Developer Studio 2006

Delphi for Microsoft Win32
Delphi for the Microsoft .NET Framework
C++Builder for Microsoft Win32
C#Builder for the Microsoft .NET Framework

For Windows

Borland®
Excellence Endures

Borland Software Corporation 100 Enterprise Way Scotts Valley, California 95066-3249 www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1997 2005 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

October 2005

PDF

Developer Studio 2006 (Common)

Getting Started

What's Developer Studio 2006? 21
What's New in Developer Studio 2006 23
Tour of the IDE 27
Starting a Project 33
Code Editor 37
Getting Started with Together 41
About Together 43
Help on Help 45

Managing the Development Life Cycle

Managing the Development Cycle Overview 49
Using Source Control 51
Using the StarTeam Integration 53
Managing Requirements with CaliberRM 57
Designing User Interfaces 59
Together Features Overview 61
Modeling Overview 63
Together Project Overview 65
Namespace and Package Overview 67
Together Diagram Overview 69
Supported UML Specifications 71
Model Element Overview 73
Annotation Overview 75
Shortcut Overview 77
Diagram Format Overview 79
Diagram Layout Overview 81
Hyperlinking Overview 83
LiveSource Overview 85
Transformation to Source Code Overview 87
OCL Support Overview 89
Patterns Overview 91
Refactoring Overview 93
Quality Assurance Facilities Overview 95
Documentation Generation Facility Overview 97
Import and Export Overview 99
Interoperability Overview 101
Compiling, Building, and Running Applications 103
Refactoring Applications 105
Refactoring Overview 107
Symbol Rename Overview (Delphi, C#, C++) 109
Extract Method Overview (Delphi) 111
Extract Resource String (Delphi) 113
Declare Variable and Declare Field Overview (Delphi) 115
Find References Overview (Delphi, C#, C++) 119
Change Parameters Overview (Delphi) 121
Sync Edit Mode (Delphi, C#, C++) 123
Undoing a Refactoring (Delphi, C#) 125
Testing Applications 127
Unit Testing Overview 129
DUnit Overview 131
NUnit Overview 135

Localizing Applications 141
Debugging Applications 143
Overview of Debugging 145
Overview of Remote Debugging 147
Deploying Applications 149

Procedures

Getting Started Procedures

Adding and Removing Files 153
Adding Components to a Form 155
Adding References 157
Adding Templates to the Object Repository 159
Configuring Together 161
Copying References to a Local Path 163
Creating a Component Template 165
Creating a Project 167
Customizing the Form 169
Customizing the Tool Palette 171
Customizing Toolbars 173
Docking Tool Windows 175
Exploring .NET Assembly Metadata 177
Exploring Windows Type Libraries 179
Finding Items on the Tool Palette 181
Installing Custom Components 183
Installing More Computer Languages 185
Renaming Files Using the Project Manager 187
Saving Desktop Layouts 189
Setting Component Properties 191
Setting Dynamic Properties 193
Setting Project Options 195
Setting Properties and Events 197
Setting The IDE To Mimic Delphi 7 199
Setting Tool Preferences 201
Using Design Guidelines with VCL Components 203
Using Online Help 205
Using To-Do Lists 207
Writing Event Handlers 209

CaliberRM Procedures

Adding a Document Reference 211
Adding a Table into a Requirement Description 213
Adding an Image to a Requirement Description 215
Assigning an Owner to a Requirement 217
Assigning Responsible Users 219
Choosing a CaliberRM Baseline 221
Choosing a CaliberRM Project 223
Creating a CaliberRM Requirement 225
Creating CaliberRM Traces 227
Deleting a CaliberRM Requirement 229
Displaying Requirement Numbers 231
Editing a Requirement 233
Editing a Requirement Description 235
Editing a Requirement Name 237

Editing Requirement Priority 239
Editing Requirement Status 241
Find a Requirement by ID 243
Launching CaliberRM Estimate Professional 245
Logging On To CaliberRM 247
Modifying CaliberRM Traceability Links 249
Moving a Requirement 251
Posting a New Requirement Discussion Message 253
Refreshing Discussion Messages 255
Replying to a Discussion Message 257
Requirement History 259
Requirement Validation 261
Specifying Requirement Comment Format 263
Updating Requirement Comments 265
Viewing a CaliberRM Project Description 267
Viewing CaliberRM Custom Tabs 269
Viewing CaliberRM Requirement Type Information 271

Compiling and Building Procedures

Building Packages 273
Finding References 275
Linking Delphi Units Into an Application 277
Previewing and Applying Refactoring Operations 279
Renaming a Symbol 281
Setting Project Options 195
Using Build Configurations 285

Debugging Procedures

Adding a Watch 287
Attaching to a Running Process 289
Debugging Remote Applications 291
Debugging VCL for .NET Source Code 293
Displaying Expanded Watch Information 295
Establishing a Connection for Remote Debugging 297
Finding References 275
Inspecting and Changing the Value of Data Elements 301
Installing a Debugger on a Remote Machine 303
Modifying Variable Expressions 305
Preparing a Project for Debugging 307
Preparing Files for Remote Debugging 309
Previewing and Applying Refactoring Operations 279
Refactoring Code 313
Renaming a Symbol 281
Resolving Internal Errors 319
Setting and Modifying Source Breakpoints 321
Setting the Search Order for Debug Symbol Tables 325
Using Tooltips During Debugging 327

Deploying Applications

Building Packages 273
Linking Delphi Units Into an Application 277

Editing Code Procedures

Creating Code Templates 333
Customizing Code Editor 335
Finding References 275

Previewing and Applying Refactoring Operations 279
Recording a Keystroke Macro 341
Refactoring Code 313
Renaming a Symbol 281
Using Bookmarks 349
Using Class Completion 351
Using Code Folding 353
Using Code Insight 355
Using Code Templates 357
Using Sync Edit 359
Using the History Manager 361

Localization Procedures

Adding Languages to a Project 365
Editing Resource Files in the Translation Manager 367
Setting the Active Language for a Project 369
Setting Up the External Translation Manager 371
Updating Resource Modules 373
Using the External Translation Manager 375

Source Control Procedures

StarTeam: Adding Files 377
StarTeam: Checking In Files 379
StarTeam: Checking Out Files 381
StarTeam: Committing Projects 383
StarTeam: Comparing File Revisions 385
StarTeam: Configuring the Integration 387
StarTeam: Editing the Active Process Item 389
StarTeam: Finding Files in the Repository 391
StarTeam: Launching the Client 393
StarTeam: Locking and Unlocking Files 395
StarTeam: Merging Source Files 397
StarTeam: Migrating Projects from the SCC Interface to the StarTeam Integration 399
StarTeam: Placing Projects and Project Groups 401
StarTeam: Pulling Projects and Project Groups 403
StarTeam: Removing Files 405
StarTeam: Reverting Files 407
StarTeam: Updating Projects 409

Together Diagram Procedures

Adding a Conditional Block 411
Adding a Member to a Container 413
Aligning Model Elements 415
Annotating a Diagram 417
Assigning an Element Stereotype 419
Associating a Lifeline with a Classifier 421
Associating a Message Link with a Method 423
Associating a Transition or a State with an Activity 425
Associating an Object with a Classifier 427
Branching Message Links 429
Browsing a Diagram with Overview Pane 431
Changing Appearance of Compartments 433
Changing Appearance of Interfaces 435
Changing Diagram Notation 437
Changing Type of a Link 439
Closing a Diagram 441

Converting Between UML 1.5 Sequence and Collaboration Diagrams 443
Copying and Pasting an Execution or Invocation Specification 445
Copying and Pasting Model Elements 447
Creating a Browse-Through Sequence 449
Creating a Deferred Event 451
Creating a Delegation Connector 453
Creating a Diagram 455
Creating a Guard Condition for a Transition 457
Creating a History Element 459
Creating a Link with Bending Points 461
Creating a Member for a State 463
Creating a Multiple Transition 465
Creating a Pin 467
Creating a Port 469
Creating a Referenced Part 471
Creating a Self-Transition 473
Creating a Sequence or Communication Diagram from an Interaction 475
Creating a Shortcut 477
Creating a Simple Link 479
Creating a Single Model Element 481
Creating a State 483
Creating a State Invariant 485
Creating an Activity for a State 487
Creating an Association Class 489
Creating an Extension Point 491
Creating an Inner Classifier 493
Creating an Internal Structure for a Node 495
Creating an Internal Transition 497
Creating Multiple Elements 499
Deleting a Diagram 501
Designing a UML 1.5 Activity Diagram 503
Designing a UML 1.5 Component Diagram 505
Designing a UML 1.5 Deployment Diagram 507
Designing a UML 1.5 Statechart Diagram 509
Designing a UML 2.0 Activity Diagram 511
Designing a UML 2.0 Component Diagram 513
Designing a UML 2.0 Deployment Diagram 515
Designing a UML 2.0 Sequence or Communication Diagram 517
Designing a UML 2.0 State Machine Diagram 519
Designing Use Case Hierarchy 521
Exporting a Diagram to an Image 523
Grouping Actions into an Activity 525
Hiding and Showing Model Elements 527
Hyperlinking Diagrams 529
Instantiating a Classifier 531
Laying Out a Diagram Automatically 533
Linking Another Interaction from an Interaction Diagram 535
Moving Model Elements 537
Printing a Diagram 539
Putting Diagram Files Under Version Control 541
Renaming a Diagram 543
Rerouting a Link 545
Resizing Model Elements 547
Searching Diagrams 549
Searching Source Code for Usages 551

Selecting Model Elements ... 553
Specifying Entry and Exit Actions ... 555
Using a Class Diagram as a View ... 557
Using Drag-and-Drop ... 559
Using Grid and Other Appearance Options ... 561
Using the UML in Color Profile ... 563
Using View Filters ... 565
Working with a Collaboration Use ... 567
Working with a Combined Fragment ... 569
Working with a Complex State ... 571
Working with a Constructor ... 573
Working with a Field ... 575
Working with a Provided or Required Interface ... 577
Working with a Relationship ... 579
Working with a Tie Frame ... 581
Working with a UML 1.5 Message ... 583
Working with a UML 2.0 Message ... 585
Working with an Instance Specification ... 587
Working with an Interface ... 589
Working with an Object Flow or a Control Flow ... 591
Working with User Properties ... 593
Zooming a Diagram ... 595

Together Documentation Generation Procedures

Configuring the Documentation Generation Facility ... 597
Generating Project Documentation ... 599

Together Object Constraint Language (OCL) Procedures

Creating a Guard Condition for a Transition ... 457
Creating a State ... 483
Creating a State Invariant ... 485
Creating an OCL Constraint ... 607
Editing an OCL Expression ... 609
Showing and Hiding an OCL Constraint ... 611
Working with a Combined Fragment ... 569

Together Pattern Procedures

Adding Participants to the Patterns as First Class Citizens ... 615
Assigning Patterns to Shortcuts ... 617
Copying and Pasting Shortcuts, Folders or Pattern Trees ... 619
Creating a Folder ... 621
Creating a Link by Pattern ... 623
Creating a Model Element by Pattern ... 625
Creating a Pattern ... 627
Creating a Shortcut to a Pattern ... 629
Creating a Virtual Pattern Tree ... 631
Deleting Patterns as First Class Citizens from the Model ... 633
Deleting shortcuts, folders or pattern trees ... 635
Editing Properties ... 637
Exporting a Pattern ... 639
Importing a Legacy Pattern ... 641
Opening the Pattern Organizer ... 643
Saving Changes in the Pattern Registry ... 645
Sharing Patterns ... 647
Sorting Patterns ... 649
Using the Pattern Organizer ... 651

Using the Pattern Registry 653
Using the Stub Implementation Pattern 655

Together Project Procedures

Activating Together Support for Projects 659
Creating a Project 661
Exporting a Project to XMI Format 663
Importing a Project Created in TCC or TAR 665
Importing a Project Created in TVS, TEC, TJB, or TPT 667
Importing a Project in IBM Rational Rose (MDL) Format 669
Importing a Project in XMI Format 671
Opening an Existing Project for Modeling 673
Sharing a Project Between TCC/TAR and Developer Studio 2006 675
Synchronizing the Model View, Diagram View, and Source Code 679
Transforming a Design Project to Source Code 681
Troubleshooting a Model 683
Working with a Namespace or a Package 685
Working with a Referenced Project 687

Together Quality Assurance Procedures

Creating a Metrics Chart 689
Exporting Audit Results 691
Printing Audit Results 693
Running Audits 695
Running Metrics 697
Viewing Audit Results 699
Viewing Metric Results 701
Working with a Set of Audits 703
Working with a Set of Metrics 705

Together Refactoring Procedures

Refactoring: "Safe Delete" 707
Refactoring: Changing Parameters 709
Refactoring: Creating Inline Variables 711
Refactoring: Extracting Interfaces 713
Refactoring: Extracting Method 715
Refactoring: Extracting Superclass 717
Refactoring: Introducing Fields 719
Refactoring: Introducing Variables 721
Refactoring: Moving Members 723
Refactoring: “Pull Members Up" and “Push Members Down” 725

Unit Test Procedures

Building Tests 727

Developer Studio 2006 for .NET

Building Applications with the ECO framework

Introduction ... 733
Overview of the ECO framework ... 735
ECO Modeling Tools Overview ... 739
Working with the ECO Service API ... 743
Working with ECO Handles ... 749
Working with ECO Subscriptions ... 755
Using State Machines with the ECO framework ... 759
Modeling Behavior with State Machines ... 761
Using Substates with the ECO framework ... 767
Object Constraint Language (OCL) and ECO Action Language ... 775
Overview of the Object Constraint Language ... 777
Using ECO Action Language ... 783
Using the ECO framework with Multi-Client Applications ... 785
The ECO framework and ASP.NET ... 787
Using the ECO Framework in Multi-Client Applications ... 789
Custom OR Mapping ... 793
Custom ECO Object-Relational Mapping Files ... 795
Custom OR Mapping with Auto-Increment Columns ... 799
Custom OR Mapping with BLOB Tables ... 801
Custom OR Mapping with Objects stored in multiple tables, with multiple keys ... 803
Custom OR Mapping with Singlelink and Compound Keys ... 805
Custom OR Mapping Using Type Discriminator Columns ... 807

Building Web Applications with ASP.NET

ASP.NET Overview ... 811
Borland DB Web Controls Overview ... 815
DB Web Controls Navigation API Overview ... 817
Working with DataViews ... 819
Working with WebDataLink Interfaces ... 821
Using DB Web Controls in Master-Detail Applications ... 823
Using XML Files with DB Web Controls ... 825
DB Web Control Wizard Overview ... 827
Deploying ASP.NET Applications ... 835

Building Web Services with ASP.NET

ASP.NET Web Services Overview ... 839
Web Services Protocol Stack ... 843
ASP.NET Web Services Support ... 845

Building Applications with Windows Forms

Windows Forms Overview ... 849
Deploying Windows Forms Applications ... 851

Building Applications with VCL.NET Components

VCL for .NET Overview ... 855
Porting VCL Applications ... 859
Language Issues in Porting VCL Applications to Developer Studio 2006 ... 861

Building Database Applications with ADO.NET

ADO.NET Overview ... 877
Borland Data Providers for Microsoft .NET ... 881
BDP.NET Data Types ... 885

BDP.NET Component Designers ... 889
Stored Procedure Overview ... 893
VCL for .NET Database Technologies ... 895
dbExpress Components overview ... 897
dbGo Components Overview ... 899
BDP Connection Pooling Overview ... 901
Getting Started with InterBase Express ... 903
Deploying Database Applications for the .NET Framework ... 909

Building Applications with Unmanaged Code

Using COM Interop in Managed Applications ... 913
Using Platform Invoke with Developer Studio 2006 ... 919
Virtual Library Interfaces ... 929
Using DrInterop ... 931
Deploying COM Interop Applications ... 933

Building Reports for .NET Applications

Using Rave Reports in Developer Studio 2006 ... 937

Procedures

ASP.NET Procedures

Adding Aggregate Values with DBWebAggregateControl ... 941
Adding Web References in ASP.NET Projects ... 943
Binding Columns in the DBWebGrid ... 947
Building an Application with DB Web Controls ... 949
Building an ASP.NET "Hello World" Application ... 951
Building an ASP.NET Application ... 953
Building an ASP.NET Database Application ... 955
Converting HTML Elements to Server Controls ... 959
Creating a Briefcase Application with DB Web Controls ... 961
Creating a Virtual Directory ... 963
Creating an XML File for DB Web Controls ... 965
Creating Metadata for a DataSet ... 967
Debugging and Updating ASP.NET Applications ... 969
Generating HTTP Messages in ASP.NET ... 971
Modifying Database Connections ... 973
Porting a Delphi for Win32 Web Service Client Application to Delphi for .NET ... 979
Setting Permissions for XML File Use ... 983
Setting Up a Cassini Web Server ... 985
Troubleshooting ASP.NET Applications ... 987
Using the ASP.NET Deployment Manager ... 991
Using the DB Web Control Wizard ... 995
Using the HTML Tag Editor ... 997
Working with ASP.NET User Controls ... 999

Database Procedures

Adding a New Connection to the Data Explorer ... 1001
Adding Aggregate Values with DBWebAggregateControl ... 941
Adding an BDP Reconcile Error dialog to your BDP Application ... 1005
Binding Columns in the DBWebGrid ... 947
Browsing a Database in the Data Explorer ... 1009
Building a Database Application that Resolves to Multiple Tables ... 1011
Building a Distributed Database Application ... 1013

Building a Windows Forms Database Application 1017
Building an Application with DB Web Controls 949
Building an ASP.NET Database Application 955
Creating a Briefcase Application with DB Web Controls 961
Creating an XML File for DB Web Controls 965
Creating Database Projects from the Data Explorer 1031
Creating Metadata for a DataSet 967
Creating Table Mappings 1035
Executing SQL in the Data Explorer 1037
Handling Errors in Table Mapping 1039
Migrating Data Between Databases 1041
Modifying Connections in the Data Explorer 1043
Modifying Database Connections 973
Passing Parameters in a Database Application 1051
Using Standard DataSets 1055
Using the Command Text Editor 1059
Using the Connection Editor Designer 1061
Using the Data Adapter Designer 1063
Using the Data Adapter Preview 1065
Using the DB Web Control Wizard 995
Using Typed DataSets 1069

ECO Framework Procedures

Adding a Derived Association End to an ECO Class Diagram 1073
Adding a Derived Attribute to an ECO Class 1075
Adding a Guard Expression to a State Transition 1077
Adding a PersistenceMapperClient to an ECO Space 1079
Adding a PersistenceMapperSharer to an ECO Space 1081
Adding a Reference to an ECO Package in a DLL 1083
Adding a Region to a State 1085
Adding a Trigger Method to an ECO Class 1087
Adding an ECO Enabled Windows Form to a Project 1089
Adding an ECO UML Package to a Project 1091
Adding an Effect to a State Transition 1093
Adding and Configuring a Connection Handle on an ECO Space 1095
Adding Columns and Nestings to an ECO Handle 1097
Adding Entry and Exit Actions to a State 1099
Adding States and Substates to an ECO State Machine Diagram 1101
Building Applications with the ECO Framework 1103
Configuring a PersistenceMapperMultiDb Component 1105
Configuring an OclVariables Component 1107
Configuring the Persistence Method of an ECO Space 1111
Converting an ECO framework Project to Developer Studio 2006 1113
Creating a New ECO Space Subclass 1115
Creating a New ECO Windows Forms Application 1117
Creating a PersistenceMapperProvider 1119
Creating an Association Class on an ECO Class Diagram 1121
Creating an ECO ASP.NET Application 1123
Creating an ECO framework State Machine Diagram 1125
Creating an ECO Package in a DLL 1127
Creating an Empty InterBase Database 1129
Creating an Event Derived Column 1131
Deploying an ECO framework Application 1133
Deriving an Attribute in Source Code 1135
Generating a Model and OR Mapping from an Existing Database 1139

Implementing a Subclass of SubscriberAdapterBase 1141
Regenerating and Updating ECO Source Code 1145
Selecting ECO UML Packages 1147
Using a Custom Object-Relational Mapping File 1149
Using the ECO Space Designer 1151
Using the Expression Editor to Build OCL and ECO Action Language Expressions 1153
Using the PersistenceMapperProvider Designer 1155

Interoperable Applications Procedures

Adding a Reference to a COM Server 1157
Adding an ActiveX Control to the Tool Palette 1159
Installing Janeva Compilers in the Tools Menu 1161

VCL for .NET Procedures

Building a VCL Forms Application 1163
Building a VCL Forms dbExpress.NET Database Application 1165
Building a VCL Forms Hello World Application 1167
Building a VCL.NET Forms ADO.NET Database Application 1169
Building an Application with XML Components 1171
Building VCL Forms Applications With Graphics 1175
Creating a New VCL.NET Component 1177
Creating Actions in a VCL Forms Application 1179
Displaying a Bitmap Image in a VCL Forms Application 1181
Drawing a Rounded Rectangle in a VCL Forms Application 1183
Drawing Rectangles and Ellipses in a VCL Forms Application 1185
Drawing Straight Lines In a VCL Forms Application 1187
Importing .NET Controls to VCL.NET 1189
Placing a Bitmap Image in a Control in a VCL Forms Application 1191
Using ActionManager to Create Actions in a VCL Forms Application 1193

Web Services Procedures

Accessing an ASP.NET "Hello World" Web Services Application 1195
Adding Web References in ASP.NET Projects 943
Building an ASP.NET "Hello World" Web Services Application 1201
Porting a Delphi for Win32 Web Service Client Application to Delphi for .NET 979

Windows Forms Procedures

Building a Windows Forms Application 1207
Building a Windows Forms Database Application 1017
Building a Windows Forms Hello World Application 1213
Building Windows Forms Menus 1215
Passing Parameters in a Database Application 1051

Developer Studio 2006 for Win32

Building Windows Applications with Win32 Forms

Windows Overview 1223

Building Web Applications with WebSnap

Win32 Web Applications Overview 1227

Building Web Services with Win32 Applications

Web Services Overview 1231

Building Database Applications for the Win32 Platform

dbGo Overview 1235
dbExpress Components 1237
BDE Overview 1239
Getting Started with InterBase Express 1241

Building Applications with VCL Components

VCL Overview 1249

Building Interoperable Applications

Building COM Applications 1255

Build Configurations

Managing C++ Build Configurations 1263

Debugging C++ Applications with CodeGuard Error Reporting

CodeGuard Overview 1265
CodeGuard Errors 1267
- Access Errors 1267
- Resource Errors 1268
- Exception Errors 1271
- Function Failure Errors 1273

CodeGuard Warnings 1275
- String Comparison Warnings 1275
- Memory Block Comparison Warnings 1275
- Pathname Merging and Splitting Warnings 1275

Building Reports for Win32 Applications

Using Rave Reports in Developer Studio 2006 1279

Procedures

CodeGuard Procedures

Using CodeGuard 1283

Database Procedures

Accessing Schema Information 1285
Configuring TSQL Connection 1287
Connecting to Databases with TDatabase 1289
Connecting to the Application Server using DataSnap Components 1291
Debugging dbExpress Applications using TSQLMonitor 1293
Executing the Commands using TSQLDataSet 1295
Fetching the Data using TSQLDataSet 1297

Managing Database Sessions Using TSession 1299
Specifying the Data to Display using TSQLDataSet 1301
Specifying the Provider using TLocalConnection or TConnectionBroker 1303
Using BDE 1305
Using DataSnap 1307
Using dbExpress 1309
Using TBatchMove 1311
Using TQuery 1313
Using TSimpleDataSet 1315
Using TSimpleObjectBroker 1317
Using TSQLQuery 1319
Using TSQLStoredProc 1321
Using TSQLTable 1323
Using TStoredProc 1325
Using TTable 1327
Using TUpdateSQL to Update a Dataset 1329

Interoperable Applications Procedures

Using COM Wizards 1331

Reporting Procedures

Adding Rave Reports to Developer Studio 2006 1333

VCL Procedures

Adding and Sorting Strings 1335
Avoiding Simultaneous Thread Access to the Same Memory 1337
Building a Multithreaded Application 1339
Building a VCL Forms "Hello world" Application 1341
Building a VCL Forms ADO Database Application 1343
Building a VCL Forms Application 1345
Building a VCL Forms Application with Decision Support Components 1347
Building a VCL Forms dbExpress Database Application 1351
Building a VCL Forms MDI Application Using a Wizard 1353
Building a VCL Forms MDI Application Without Using a Wizard 1355
Building a VCL Forms SDI Application 1359
Building a VCL Forms Web Browser Application 1361
Building a Windows "Hello World" Application 1363
Building a Windows Application 1365
Building an Application with XML Components 1367
Building Application Menus 1371
Building VCL Forms Applications With Graphics 1373
Copying a Complete String List 1375
Copying Data From One Stream To Another 1379
Creating a New VCL Component 1381
Creating a VCL Form Instance Using a Local Variable 1383
Creating a VCL Forms ActiveX Active Form 1385
Creating Actions in a VCL Forms Application 1387
Creating Strings 1389
Defining the Thread Object 1391
Deleting Strings 1395
Displaying a Bitmap Image in a VCL Forms Application 1399
Displaying a Full View Bitmap Image in a VCL Forms Application 1401
Displaying an Auto-Created VCL Form 1403
Drawing a Polygon in a VCL Forms Application 1405
Drawing a Rounded Rectangle in a VCL Forms Application 1407
Drawing Rectangles and Ellipses in a VCL Forms Application 1409

Drawing Straight Lines In a VCL Forms Application 1411
Dynamically Creating a VCL Modal Form 1413
Dynamically Creating a VCL Modeless Form 1415
Handling Exceptions 1417
Initializing a Thread 1419
Iterating Through Strings in a List 1421
Placing A Bitmap Image in a Control in a VCL Forms Application 1425
Reading a String and Writing It To a File 1427
Renaming Files 1429
Using ActionManager to Create Actions in a VCL Forms Application 1431
Using the Main VCL Thread 1433
Waiting for Threads 1435
Writing Cleanup Code 1439
Writing the Thread Function 1441

WebSnap Procedures

Building a WebSnap "Hello World" Application 1443
Building a WebSnap Application 1445
Debugging a WebSnap Application using the Web Application Debugger 1447

Concepts

General

Getting Started

The Developer Studio 2006 integrated development environment (IDE) provides many tools and features to help you build powerful applications quickly. Not all features and tools are available in all editions of Developer Studio 2006. For a list of features and tools included in your edition, refer to the feature matrix on http://www.borland.com/delphi.

In This Section

What's Developer Studio 2006?
Provides a product overview and describes the Developer Studio 2006 tools for managing the development life cycle.

What's New in Developer Studio 2006
Introduces key new features and functionality in the product.

Tour of the IDE
Describes the various IDE elements.

Starting a Project
Describe the parts of a project and provides a list of projects supported in Developer Studio 2006.

Code Editor
Describes the features of the Developer Studio 2006 Code Editor.

Help on Help
Explains how information is organized in the online Help and lists additional developer resources.

Procedures

What's Developer Studio 2006?

Developer Studio 2006 is an integrated development environment (IDE) for building Delphi, Delphi for .NET, C#, and C++ applications. The Developer Studio 2006 IDE provides a comprehensive set of tools that streamline and simplify the development life cycle. The tools available in the IDE depend on the edition of Developer Studio 2006 you are using. The following sections briefly describe these tools.

Defining Requirements

Developer Studio 2006 provides an interface to Borland CaliberRM, a Web-based requirements definition and management system designed to help control the product development process. Within the IDE, you can access CaliberRM to collaborate on project requirements and ensure that your applications meets end-user needs.

Modeling Applications

Modeling can help you can improve the performance, effectiveness, and maintainability of your applications by creating a detailed visual design before you ever write a line of code. Developer Studio 2006 provides UML-based class diagramming tools and a framework of Enterprise Core Objects (ECO) to help you create model-powered .NET applications.

Designing User Interfaces

The Developer Studio 2006 visual designer surface lets you create graphical user interfaces by dragging and dropping components from the **Tool Palette** to a form. Using the designers, you can create VCL Forms, Windows Forms, Web Forms, and HTML pages.

Generating and Editing Code

Developer Studio 2006 auto-generates much of your application code as soon as you begin a project. To help you complete the remaining application logic, the text-based **Code Editor** provides features such as refactoring, synchronized editing, code completion, recorded keystroke macros, and custom key mappings. Syntax highlighting and code folding make your code easier to read and navigate.

Compiling, Debugging, and Deploying Applications

Within the IDE, you can set compiler options, compile and run your application, and view compiler messages. The integrated Borland .NET and Borland Win32 debuggers help you find and fix runtime and logic errors, control program execution, and step through code to watch variables and modify data values. The Developer Studio 2006 ASP.NET Deployment Manager can assist you in copying the files required by your ASP.NET application to a web server. Additionally, the .NET Framework includes several utilities to help you prepare applications for deployment. Developer Studio 2006 includes InstallShield Express for creating Windows Installer setups.

Controlling Access and Tracking Changes to Code

Source control systems enable team development by controlling access and tracking changes to source code and other files. Developer Studio 2006 provides a full-featured, direct integration with StarTeam, Borland's automated change and software configuration management system. Within the Developer Studio 2006 IDE, you can perform common source control tasks, such as file check in, check out, and synchronization.

The .NET Framework

The Microsoft .NET Framework provides the foundation for building and running .NET applications. The Framework includes the common language runtime and class library. The common language runtime manages the execution of code and provides services, such as memory management and cross-language integration, that simplify the development process. The class library is a collection of reusable, object-oriented components for developing .NET applications that take advantage of the common language runtime services.

Developer Studio 2006 makes the entire Framework class library available in the IDE to help you develop .NET applications. Developer Studio 2006 enhances the Framework in the following areas:

- The Developer Studio 2006 Borland Data Providers for .NET provide access to InterBase, Oracle, DB2 Universal, and Microsoft SQL Server databases.
- Several database utilities assist in performing tasks such as connecting to databases, browsing and editing databases, and executing SQL queries.
- The .NET Menu Designers simplify the creation of main menus and context menus on Windows Forms.

What's New in Developer Studio 2006

Developer Studio 2006 provides key new features for developing Delphi, Delphi for .NET, C#, and C++ applications

C++ Personality

Developer Studio 2006 provdes support for developing C++ applications. The following key features are available for only the C++ personality:

- **Build configurations:** You can create and quickly switch between multiple build configurations, which store sets of command-line options for build tools such as the compiler and linker.
- **Build events:** You can specify commands to execute at certain points in the build by righ-clicking a buildable file in the **Project Manager** and choosing **Build Events**.
- **CodeGuard integration:** CodeGuard, a tool that provides runtime debugging for C++ applications, has tighter integration with Developer Studio 2006.
- **Dinkumware runtime libraries:** Dinkumware runtime libraries are provided for enhanced conformance to ANSI/ISO C++ standards.

IDE

- **New Memory Manager:** This relncludes a new memory manager that significantly improves start-up time, runtime speed, and hyperthreading performance.
- **Improved Speed for Several Features:** The Search | Find Uses/Import Namespace, Find Class, and Change Parameters features all have significant performance improvements in this release.
- **Change Parameters Refactoring:** You can add, remove, or change the ordering of method parameters using this refactoring. Change parameters refactoring is available for Delphi for Win32 and Delphi for .NET.
- **Message view:** The **Message** view automatically scrolls to display new items.
- **Project Repository improvements:** You can now add a starter project, demo, template, or other frequently used file to the **Object Repository**, which causes it to become available on the **New** menu.

Form Designer

- **Design Guidelines:** When you move components on a form, design guidelines appear and help you align components.
- **Form Positioner:** This new view appears in the lower-right corner of the **Form Designer**. You can expand this view and quickly reposition the runtime position of the form.

Code Editor

- **New code templates:** Code templates provide a means of automating the task of typing frequently used code structures. Developer Studio 2006 provides a library of templates for every supported language, and you can add other new templates by choosing **File** ▸ **New** ▸ **Other** ▸ **Other Files** ▸ **Code Template**.
- **Surround templates:** You can right-click a selected a block of code and choose **Surround** to view a list of possible templates with which to surround your code.
- **Live templates editing:** When you add a code template to your source code, you can TAB through fields and insert points to quickly populate the template with logic.

- **Block completion:** Block closures are automatically added as needed when you edit code.
- **Method navigation:** You can quickly navigate between methods in your source code using a series of hotkeys. CTRL+ALT+UP and CTRL+ALT+DOWN move to the previous and next method, respectively. CTRL+ALT+HOME and CTRL+ALT+END move to the first and last methods in the source, respectively. CTRL+ALT+Q^L toggles class lock, which causes method navigation to apply to only the current class.
- **Improved Code Editor gutter:** The **Code Editor** gutter is now more readable and less cluttered.
- **Diff highlighting:** Yellow highlights appear in the **Code Editor** gutter next to lines modified since your last save. Green highlights appear next to lines that have been modified and saved in the current editing session.
- **Close all other pages:** You can close all other pages by right-clicking a page tab and choosing **Close All Other Pages**.

Debugger

- **Remote debugging:** Remote debugging is now available for native Win32 applications, managed applications, and ASP.NET applications.
- **Symbol table management:** You can now specify the order in which symbols tables are loaded for a particular module that you are debugging. You can also limit the search to specific symbol tables, which can speed up the debugging process.
- **Expandable watches:** You can now inspect the values of members within a watched object, as well as elements within an object. Expanded tooltips are available for watched objects.
- **CPU view:** In the **CPU** view, you can now select multiple items and copy them to the clipboard..
- **Sort by load order:** In the **Module** view, modules can now be sorted by their load order.
- **Close implicitly opened files:** The debugger now closes any files it automatically opens in a debugging session.

ECO Framework

- **ECO State Machines:** The addition of ECO state machine diagrams allow you to model the behavior of classes. ECO state machine diagrams support entry and exit actions, transition effects, OCL guard expressions, and concurrent state machines.
- **ECO Action Language:** ECO Action Language is an extension of the Object Constraint Language (OCL) that allows side-effects. You can use ECO Action Language on state machine diagrams to completely specify behavior on the diagram itself, rather than writing code.
- **OCL Expression Editor:** The OCL Expression Editor is now available from both the ECO WinForm designer, and on ECO UML diagrams.
- **Reverse and Wrap an Existing Database with ECO:** The ECO space designer now contains a tool to help you create an ECO model from an existing database. This wizard steps you through the process of selecting a database and customizing the OR mapping.

Modeling

- **Together UML Tools:** New diagram types and code constructs such as interfaces, enumerations and structures may be created from the **Model** view. The following diagrams and constructs are available: Class diagram, Use case diagram, Sequence diagram, Collaboration diagram, State chart diagram, Activity diagram, Component diagram, Deployment diagram, Class, Interface, Structure, Enumeration, Delegate, Namespace, Object, Constraint, Note.

- **Together Engine:** The core engine has been rewritten to provide increased speed and stability

ASP.NET Web Development

- **Show referenced assemblies:** The **Deployment Manager** can now show all assemblies referenced by the current project.
- **Adding external files:** You can easily choose the external files that you want to deploy using the **External Files** dialog box.
- **Markup source preservation:** When you edit a markup document using the MSHTML control, the IDE now preserves whitespace, user-specified tag and attribute formatting, and closing tags.
- **Change default layout:** You can now change the default layout in the **Design Editor** to be **Grid Layout** or **Flow Layout**. Choose **Tools** ▸ **Options** ▸ **HTML/ASP.NET Options** to change the default layout.
- **Cassini:**Developer Studio 2006 provides better support for the Cassini debugging web server. A pre-built server is included with the IDE.

Database

Many changes have been made to improve support for database application development in Developer Studio 2006.

dbExpress

- **dbExpress Unicode support:** The MSSQL driver now supports unicode.
- **ConnectionString property:** The ConnectionString property in dbExpress lets you pass all database options and connection information using a single connection string.
- **Customizable decimal separator:** You can now specify the decimal separator.
- **MSSQL Return values:** Support for dbExpress MSSQL return values from Stored Procedures has been added.
- **TSQLQuery support:** Support for TQSLQuery OUT and INOUT parameters has been added.

BDP.NET Updates

- **Connection pooling support:** You can now use connection pooling to decrease connection time by using a connection from an existing pool. Connection pooling options are available on the **Connections Editor** dialog box.
- **Reconcile Error Dialog:** When an error occurs during a database Delete, Insert or Update operation, the **Reconcile Error** dialog box lets you to decide which data source to use, whether to abort the operation completely or to continue on with the next update.

General database features

- Support for MySQL 4.0.24 BDP Provider
- Customizable SQL type mapping for Data Migration
- QuoteObjects support for CREATE/ALTER/DROP in ISQLSchemaCreate
- Related Objects and ForeignKey support in ISQLExtendedMetaData (for ORACLE, Interbase, MSSQL, and Sybase)

- Support for the following Oracle 9i data types: `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL YEAR TO MONTH`, and `INTERVAL DAY TO SECOND`

VCL`

New components: The following new components have been added to the Visual Component Library:

- TTrayIcon
- TGridPanel
- TFlowPanel

New classes: The following new classes have been added:

- TCustomTransparentControl
- TMargins
- TPadding

Delphi Language Enhancements

Records: The following support has been added for record types:

- Operator overloading
- Non-virtual method declaration
- Regular instance methods
- Constructors with non-empty parameter lists
- Static methods and properties

Note: Destructors in records are not permitted.

StarTeam Integration

- **Search:** The StarTeam integration has been enhanced to include access to the Borland Search feature.
- **Visual Diff / Merge:** You can now use the **Visual Diff** and **Visual Merge** features from the embedded StarTeam client, even if you do not have the standalone StarTeam client installed.

CaliberRM Integration

- **Integration:** You can log on to the CaliberRM server directly from the IDE. Once you are logged on, you can use the CaliberRM requirements management features to add, remove, or update requirements.
- **Linking requirements to source code files:** You can link a requirement directly to source code files within the IDE.

Tour of the IDE

When you start Developer Studio 2006, the integrated development environment (IDE) launches and displays several tools and menus. The IDE helps you visually design user interfaces, set object properties, write code, and view and manage your application in various ways.

The default IDE desktop layout includes some of the most commonly used tools. You can use the **View** menu to display or hide certain tools. You can also customize and save the desktop layouts that work best for you.

The tools available in the IDE depend on the edition of Developer Studio 2006 you are using and include the

- Welcome Page
- Accessibility Options
- Forms
- Form Designer
- Tool Palette
- Object Inspector
- Object Repository
- Project Manager
- Data Explorer
- Structure View
- History Manager
- Code Editor

The following sections describe each of these tools.

Welcome Page

When you open Developer Studio 2006, the **Welcome Page** appears with a number of links to developer resources, such as product-related articles, training, and online Help. As you develop projects, you can quickly access them from the list recent projects at the top of the page. If you close the **Welcome Page**, you can reopen it by choosing **View** ▸ **Welcome Page**.

Accessibility Options

The IDE's main menu supports MS Active Accessibility (MSAA). This means that you can use the Windows accessibility tools from the Start Menu via**All Programs** ▸ **Accessories** ▸ **Accessibility**.

Forms

Typically, a form represents a window or HTML page in a user interface. At design-time, a form is displayed on the Designer surface. You add components from the **Tool Palette** to a form to create your user interface.

Developer Studio 2006 provides several types of forms, as described in the following sections. Select the form that best suits your application design, whether it's a Web application that provides business logic functionality over the Web, or a Windows application that provides processing and high-performance content display. To switch between the Designer and **Code Editor**, click their associated tabs below the IDE.

To access forms, choose **File** ▸ **New** ▸ **Other**.

Windows Forms

Use Windows Forms to build native Windows applications that run in a managed environment. You use the .NET classes to build Windows clients which presents two major advantages—it allows application clients to use features unavailable to browser clients, and it leverages the .NET Framework infrastructure. Windows Forms present a programming model that takes advantage of a unified .NET Framework (for security and dynamic application updates, for instance) and the richness of GUI Windows clients. You use Windows controls, such as buttons, list boxes, and text boxes, to build your Windows applications.

To access a Windows Form, choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **Windows Forms Application**.

ASP.NET Web Forms

Use ASP.NET Web Forms to create applications that can be accessed from any Web browser on any platform. You use the .NET classes to create a ASP.NET Web Forms application. The form consists of the visual representation of the HTML, the actual HTML, and a code-behind file.

To access an ASP.NET Web Form, choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **ASP.NET Web Application**.

VCL Forms

Use VCL Forms to create applications that use VCL.NET components to run in the .NET Framework. You use the Borland Visual Component Library for .NET classes to create a VCL Forms application.

VCL Forms are especially useful if you want to port an existing Delphi application containing VCL controls to the .NET environment, or if you are already familiar with the VCL and prefer to use it.

To access a VCL Forms, choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.

Form Designer

The **Form Designer**, or Designer, is displayed automatically when you are using a form. The appearance and functionality of the Designer depends on the type of form you are using. For example, if you are using an ASP.NET Web Form, the Designer will display an HTML tag editor. To access the Designer, click the **Design** tab at the bottom of the IDE.

Visual Components

Visual components appear on the form at design-time and are visible to the end user at runtime. They include such things as buttons, labels, toolbars, and listboxes.

Form Preview

A preview icon at the bottom right of the Designer (for VCL Forms) shows the positioning of your form as it will appear on the screen at runtime. This allows you to position the forms of your application in relation to each other as you design them.

HTML Designer

Use the **HTML Designer** to view and edit ASP.NET Web Forms or HTML pages. This Designer provides a **Tag Editor** for editing HTML tags alongside the visual representation of the form or page. You can also use the **Object Inspector** to edit properties of the visible items on the HTML page and to display the properties of any current HTML tag in the **Tag Editor**. A combo box located above the **Tag Editor** lets you display and edit SCRIPT tags.

To create a new HTML file, choose **File** ▸ **New** ▸ **Other** ▸ **Web Documents** ▸ **HTML Page**.

Nonvisual Components and the Component Tray

Nonvisual components are attached to the form, but they are only visible at design-time; they are not visible to end users at runtime. You can use nonvisual components as a way to reuse groups of database and system objects or isolate the parts of your application that handle database connectivity and business rules.

When you add an nonvisual component to a form, they are displayed in the component tray at the bottom of the Designer surface. The component tray lets you distinguish between visual and nonvisual components.

Design Guidelines

If you are creating components for a form, you can register an object type and then indicate various points on or near a component's bounds that are "alignment" points. These "alignment" points are vertical or horizontal lines that cut across a visual control's bounds.

When you have the alignment points in place, you can supply UI guideline information so that each component will adhere to rules such as distance between controls, shortcuts, focus labels, tab order, maximum number of items (listboxes, menus), etc. In this way, the Form Designer can assist the Code Developer in adhering to established UI guidelines.

If the Snap to Grid option is enabled, and Use Designer Guidelines is also enabled, the designer guidelines will take precedence. This means that if a grid point is within the tolerance of the new location and a guideline is also within that distance away, then the control will snap to the guideline instead of the grid position, even if the guideline does not fall on the grid position. The snap tolerance is determined by the grid size. Even if the Snap to Grid and Show Grid options are disabled, the Designer will still use the grid size in determining the tolerance.

This feature is currently only available in VCL and VCL.NET only (This includes C++). Winforms does not yet have this feature. See the link at the end of this topic for more information about setting Designer Guidelines.

Tool Palette

The **Tool Palette** contains items to help you develop your application. The items displayed depend on the current view. For example, if you are viewing a form on the Designer, the **Tool Palette** displays components that are appropriate for that form. You can double-click a control to add it to your form. If you are viewing code in the **Code Editor**, the **Tool Palette** displays code segments that you can add to your application.

Customized Components

In addition to the components that are installed with Developer Studio 2006, you can add customized or third party components to the **Tool Palette** and save them in their own category.

Component Templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, you can save them as a component template. Later, by selecting the template from the **Tool Palette**, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time. You can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

Object Inspector

The **Object Inspector** lets you set designtime properties and create event handlers for components. This provides the connection between your application’s visual appearance and the code that makes the application run. The **Object Inspector** contains two tabs: **Properties** and **Events**.

Use the **Properties** tab to change physical attributes of your components. Depending on your selection, some category options let you enter values in a text box while others require you to select values from a drop-down box.

For Boolean operations, you toggle between True or False. After you change your components' physical attributes, you create event handlers that control how the components function.

Use the **Events** tab to specify the event of a specific object you select. If there is an existing event handler, use the drop-down box to select it. By default, some options in the **Object Inspector** are collapsed. To expand the options, click the plus sign (+) next to the category.

Certain nonvisual components, for example, the Borland Data Providers, allow quick access to editors such as the **Connection Editor** and **Command Text Editor**. You can access these editors in the **Designer Verb** area at the bottom of the **Object Inspector**. To open the editors, point your cursor over the name of the editor until your cursor changes into a hand and the editor turns into a link. Alternatively, you can right-click the nonvisual component, scroll down to its associated editor and select it. Note that not all nonvisual components have associated editors. In addition to editors, this area can also display hyperlinks to show custom component editors, launch a web page and show dialog boxes.

Object Repository

To simplify development, Developer Studio 2006 offers pre-designed templates, forms, and other items that you can access and use in your application.

Inside the Object Repository

The **Object Repository** contains items that address the types of applications you can develop. It contains templates, forms, and many others items. You can create projects such as class library, control library, console applications, HTML pages, and many others by accessing the available templates.

The **Object Repository** is accessible by choosing **File** ▸ **New** ▸ **Other**. A **New Items** dialog box appears, displaying the contents of the **Object Repository** . You can also edit or remove existing objects from the **Object Repository** by right-clicking the **Object Repository** to view your editing options.

Object Repository Templates

You can add your own objects to the **Object Repository** as templates to reuse or share with other developers. Reusing objects lets you build families of applications with common user interfaces and functionality to reduce development time and improve quality.

You can add a starter project, demo, template, or other useful file to the Repository, and then make it available through the **New** menu. Choose **Project** ▸ **Add to Repository**. Select your file. Now when you select the **File New** command, you can choose the file you just added and work with a new copy of it.

Project Manager

A project is made up of several application files. The **Project Manager** lets you view and organize your project files such as forms, executables, assemblies, objects and library files. Within the **Project Manager**, you can add, remove, and rename files. You can also combine related projects to form project group, which you can compile at the same time.

Add References

You can integrate your legacy COM servers and ActiveX controls into managed applications by adding references to unmanaged DLLs to your project, and then browse the types just as you would with managed assemblies. Choose **Project** ▸ **Add Reference** to integrate your legacy COM servers or ActiveX controls. Alternatively, right-click the **Reference** folder in the **Project Manager** and click **Add Reference**. You can add other .NET assemblies, COM/ ActiveX components, or type libraries using the **Add Reference** feature.

Copy References to a Local Path

During runtime, assemblies must be in the output path of the project or in the Global Assembly Cache (GAC) for deployment. In the **Project Manager**, you can right-click an assembly and use the **Copy Local** setting to copy the reference to the local output path. Follow these guidelines to determine whether a reference must be copied.

- If the reference is to an assembly created in another project, select the **Copy Local** setting.
- If the assembly is in the GAC, do not select the **Copy Local** setting.

Add Web References

You can quickly add a Web Reference to your client application and access the Web Service you want to use. When you add a Web Reference, you are importing a WSDL document into your client application, which describes a particular Web Service. Once you imported the WSDL document, Developer Studio 2006 generates all the interfaces and class definitions you need for calling that Web Service. To use the Add Web Reference feature, from your **Project Manager**, right-click the **Web Services** node.

Data Explorer

The **Data Explorer** lets you browse database server-specific schema objects including tables, fields, stored procedure definitions, triggers, and indexes. Using the context menus, you can create and manage database connections. You can also drag and drop data from a data source to most forms to build your database application quickly.

Structure View

The **Structure View** shows the hierarchy of source code or HTML displayed in the **Code Editor**, or components displayed on the Designer. When displaying the structure of source code or HTML, you can double-click an item to jump to its declaration or location in the **Code Editor**. When displaying components, you can double-click a component to select it on the form.

If your code contains syntax errors, they are displayed in the **Errors** folder in the **Structure View**. You can double-click an error to locate its source in the **Code Editor**.

You can control the content and appearance of the **Structure View** by choosing **Tools** ▸ **Options** ▸ **Environment Options** ▸ **Explorer** and changing the settings.

History Manager

The **History Manager** lets you see and compare versions of a file, including multiple backup versions, saved local changes, and the buffer of unsaved changes for the active file. If the current file is under version control, all types of revisions are available in the **History Manager**. The **History Manager** is displayed to the right of the **Code** tab and contains the following tabbed pages:

- The **Contents** page displays current and previous versions of the file.
- The **Diff** page displays differences between selected versions of the file.
- The **Info** page displays all labels and comments for the active file.

You can use the **History Manager** toolbar to refresh revision information, revert a selected version to the most current version, and synchronize scrolling between the source viewers in the **Contents** or **Diff** pages and the **Code Editor**.

Code Editor

The **Code Editor** provides a convenient way to view and modify your source code. It is a full-featured, customizable, UTF8 editor that provides refactoring, automatic backups, Code Insight, syntax highlighting, multiple undo capability, context-sensitive Help, Code Templates, Smart Block Completion, Find Class, Find Unit/Import Namespace, and more. Choose the Code Editor link in the section below to view descriptions for each of these Code Editor features.

Starting a Project

A project is a collection of files that is used to create a target application. This collection of files consists of the files you include and modify directly, such as source code files and resources, and other files that Developer Studio 2006 maintains to store project settings, such as the .bdsproj project file. Projects are created at design time, and they produce the project target files (.exe, .dll, .bpl, etc.) when you compile the project.To assist in the development process, the **Object Repository** offers many pre-designed templates, forms, files, and other items that you can use to create applications.

To create a project, click **New** from the **Welcome Page** and select the type of application you want to create, or choose **File** ▸ **New** ▸ **Other**. To open an existing project, click **Project** from the **Welcome Page** or choose **File** ▸ **Open Project**.

This section includes information about

- Types of projects
- Working with unmanaged code

Type of Projects

Depending on the edition of Developer Studio 2006 that you are using, you can create traditional Windows applications, ASP.NET Web applications, ADO.NET database applications, Web Services applications, and many others. Developer Studio 2006 also supports assemblies, custom components, multi-threading, and COM. For a list of the features and tools included in your edition, refer to the feature matrix on either the Borland Delphi web page or the Borland C#Builder web page.

Windows Applications

You can create Windows applications using Windows Forms to provide processing and high-performance content display. In addition to traditional uses for Windows applications, a Windows application can be used with constructs from the newer .NET framework. For instance, a Windows application can function as a front end to an ADO.NET database.

ASP.NET Web Applications

You can create Web applications using ASP.NET Web Forms to provide Web access to databases and Web Services. Web Forms provide the user interface for Web applications and consist of HTML, server controls, and application logic in code-behind files. Developer Studio 2006 lets you drag and drop components and provides in-place HTML editing.

In addition to drag and drop components and visual designers, Borland provides an easy way to create application menus and submenus. The .NET Menu Designers MainMenu and ContextMenu are components that work like editors to let you visually design menus and quickly code their functionality.

ASP.NET Web Services Applications

You can create Web Services applications that deliver content, such as HTML pages or XML documents, over the Web. Web Services is an internet-based integration methodology that allows applications to connect through the Web and exchange information using standard messaging protocols.

Developer Studio 2006 simplifies the creation of Web Services by providing methods for creating a SOAP Server application. The .asmx and .dll files are created automatically and you can test the Web Service within the IDE, without writing a client application for it.

When writing a client application that uses, or *consumes*, a published Web Service, you can use the UDDI Browser to locate and import WSDL that describes the Web Service into your client application.

VCL.NET Applications

You can use VCL Forms to create a .NET Windows application that uses components from the VCL.NET framework.

Developer Studio 2006 simplifies the task of building .NET-enabled applications by supporting VCL components that have been augmented to run on the .NET Framework. This eliminates the need for you to create custom components to provide standard VCL component capabilities. This makes the process of porting Win32 applications to .NET much simpler and more reliable.

Database Applications

Whether your application uses Windows Forms, Web Forms, or VCL Forms, Developer Studio 2006 has several tools that make it easy to connect to a database, browse and edit a database, execute SQL queries, and display live data at design time.

The ADO.NET framework data providers let you access MS SQL, Oracle, and ODBC and OLE DB-accessible databases. The Borland Data Providers (BDP.NET) let you access MS SQL, Oracle, DB2, and InterBase databases. You can connect to any of these data sources, expose their data in datasets, and use SQL commands to manipulate the data. Using BDP.NET provides the following advantages:

- Portable code that's written once and connects to any supported database.
- Open architecture that allows you to provide support for additional database systems.
- Logical data types that map easily to .NET native types.
- Consistent data types that map across databases, where applicable.
- Unlike OLE DB, there is no need for a COM/Interop layer.

When using VCL Forms and the VCL.NET framework components, you can extend database support even further by using the BDE.NET, dbExpress.NET, and Midas Client for .NET connection technologies.

Model-Driven Applications

Modeling is a term used to describe the process of software design. Developing a model of a software system is roughly equivalent to an architect creating a set of blueprints for a large development project. Like a set of blueprints, a model not only depicts the system as a whole, but also allows you to focus in on specifics such as structural and behavioral details. Abstracted away from any particular programming language (and at some levels, even from specific technology), the model allows all participants in the development cycle to communicate in the same language.

Borland's Model Driven Architecture (MDA) describes an approach to software engineering where the modeling tools are completely integrated within the development environment itself. The MDA is designed around Borland’s Enterprise Core Objects (ECO) framework. The ECO framework is a set of interface, classes, and custom attributes that provide the communication conduit between your application and the modeling-related features of the IDE.

The ECO features include:

- Automatic mapping of the model classes, with their attributes and relationships, to a relational schema.
- Automatic evolution of schema when the model changes.
- Specification of the persistence backend. You can choose to store objects in a relational database or in an XML file.
- Design-time structural validation of the model and its Object Constraint Language (OCL) expressions.
- Runtime validation of the OCL expressions.

- An event mechanism that allows you to receive notifications whenever objects are added, changed, or removed.

Developer Studio 2006 IDE leverages the ECO framework to provide an integrated surface on which to develop your application model. The IDE and its modeling surface features include:

- Creating model-driven applications as a new kind of project.
- Creating class diagrams, and manipulating model elements (packages, and classes) directly on the surface.
- Adding, removing, and changing class attributes and methods on the class diagram.
- Two-way updating between source code and the modeling surface. Changes in source code are reflected in the graphical depiction, and vice versa.
- Two-way navigating between model elements and source code. You can navigate from the graphical depiction of a model element directly to its corresponding source code. Similarly, you can navigate from a modeled class in source code directly to its graphical diagram on the modeling surface.
- Exporting and importing models using XMI 1.1.

Note: Not all modeling features are available in all editions of Developer Studio 2006. To determine the modeling features supported in your product edition, refer to the feature matrix on either the Borland Delphi web page or the Borland C#Builder web page.

Assemblies

An assembly is a logical package, much like a DLL file, that consists of manifests, modules, portable executable (PE) files, and resources (.html, .jpeg, .gif) and is used for deployment and versioning. An application can have one or more assemblies that are referenced by one or more applications, depending on whether the assemblies reside in an application directory or in a global assembly cache (GAC).

Additional Projects

In addition to the project types described above, Developer Studio 2006 provides templates to create class libraries, control libraries, console applications, Visual Basic applications, reports, text files, and more. These templates are stored in the **Object Repository** and you can access them by choosing **File** ▸ **New** ▸ **Other**.

Unmanaged Code and COM/Interop

Unmanaged code refers to applications that do not target the .NET Framework Common Language Runtime (CLR). COM/Interop is a .NET service that allows seamless interoperation between managed and unmanaged code. The COM/Interop service allows you to leverage existing COM servers and ActiveX controls in your .NET applications, and expose .NET components in legacy unmanaged applications. The Developer Studio 2006 IDE includes tools to help you integrate your legacy COM servers and ActiveX controls into managed applications. Additionally, you can add references to unmanaged DLLs to your project, and then browse the types contained, just as you would with managed assemblies.

Code Editor

The **Code Editor** is a full-featured, customizable, UTF8 editor that provides syntax highlighting, multiple undo capability, and context-sensitive Help for language elements.

As you design the user interface for your application, Developer Studio 2006 generates the underlying code. When you modify object properties, your changes are automatically reflected in the source files.

Because all of your programs share common characteristics, Developer Studio 2006 auto-generates code to get you started. You can think of the auto-generated code as an outline that you can examine to create your program.

Note: If you are using WinForms, do not modify the auto-generated code for the *InitializeComponents* method. Doing so will cause your form to disappear when you click the **Design** tab.

The **Code Editor** provides the following features to help you write code:

- Change Bars
- Code Insight
- Sync Edit
- Code Completion
- Code Browsing
- Help Insight
- Code Templates
- Code Folding
- To-Do Lists
- Keystroke Macros
- Bookmarks
- Block comments

Change Bars

When you make changes to your code with the **Code Editor** in Developer Studio 2006, the left margin of the **Code Editor** will display a yellow change bar to indicate that changes have been made after the last **Save** operation. You can customize the change bars to display in other colors.

Code Insight

Code Insight refers to a subset of features embedded in the **Code Editor** (such as Code Parameter Hints, Code Hints, Help Insight, Code Completion, Class Completion, Block Completion, and Code Browsing) that aid in the code writing process. These features help identify common statements you wish to insert into your code, and assist you in the selection of properties and methods. Some of these features are described in more detail in the sub-sections below.

To invoke **Code Insight**, press CTRL+SPACE while using the **Code Editor**. A pop-up window displays a list of symbols that are valid at the cursor location.

To enable and configure **Code Insight** features, choose **Tools** ▸ **Options** and click **Code Insight**.

When you're using the Delphi Language, the pop-up window filters out all interface method declarations that are referred to by property read or write clauses. The window displays only properties and stand-alone methods declared in the interface type. Code insight supports WM_xxx, CM_xxx, and CN_xxx message methods based on like named constants from all units in the uses clause.

Code Parameter Hints

Displays a hint containing argument names and types for method calls. Available between the parenthesis of a call i.e. ShowMessage (|);

You can invode Code Parameter Hints by pressing CTRL+SHIFT+SPACE.

Code Hints

Display a hint containing information about the symbol such as type, file and line # declared at.

You can display Code Hints by hovering the mouse over an identifier in your code, while working in the Code Editor.

Note: Code Hints only work when you have disabled the Help Insight feature.

Help Insight

Help Insight displays a hint containing information about the symbol such as type, file, line # declared at, and any XML documentation associated with the symbol (if available).

Invoke Help Insight by hovering the mouse over an identifier in your code, while working in the Code Editor. You can also invoke Help Insight by pressing CTRL+SHIFT+H.

Code Completion

The Code Completion feature displays a drop-down list of available symbols at the current cursor location. You invoke Code Completion for your specific language in the following way:

Delphi — CTRL + SPACE + .

C# — CTRL + SPACE + .

C++ — CTRL + SPACE + —>

Class Completion

Class completion simplifies the process of defining and implementing new classes by generating skeleton code for the class members that you declare. By positioning the cursor within a class declaration in the interface section of a unit and pressing CTRL+SHIFT+C, any unfinished property declarations are completed. For any methods that require an implementation, empty methods are added to the implementation section. They are also on the editor context menu.

Block Completion

When you press ENTER while working in the Code Editor and there is a block of code that is incorrectly closed, the Code Editor enters the closing block token at the next available empty line after the current cursor position. For instance, if you are using the **Code Editor** with the Delphi language, and you type the token **begin** and then press ENTER, the Code Editor automatically completes the statement so that you now have: **begin end;** . This feature also works for the C# and C++ languages.

Code Browsing

While using the **Code Editor** to edit a VCL Form application, you can hold down the CTRL key while passing the mouse over the name of any class, variable, property, method, or other identifier. The mouse pointer turns into a hand and the identifier appears highlighted and underlined; click on it, and the **Code Editor** jumps to the declaration of the identifier, opening the source file, if necessary. You can do the same thing by right-clicking on an identifier and choosing **Find Declaration**.

Code browsing can find and open only units in the project Search path or Source path, or in the product Browsing or Library path. Directories are searched in the following order:

1. The project Search path (**Project** ▸ **Options** ▸ **Directories/Conditionals**).
2. The project Source path (the directory in which the project was saved).
3. The global Browsing path (**Tools** ▸ **Options** ▸ **Library**).
4. The global Library path (**Tools** ▸ **Options** ▸ **Library**).

The Library path is searched only if there is no project open in the IDE.

Code Navigation

The sections below describe features that you can use to navigate through your code while you are using the **Code Editor**.

Method Hopping

You can navigate between methods using a series of editor hotkeys. You can also lock the hopping to occur only within the methods of the current class. For example, if class lock is enabled and you are in a method of TComponent, then hopping is only available within the methods of TComponent.

The keyboard shortcuts for Method Hopping are as follows:

- CTRL+Q^L - toggles class lock
- CTRL+ALT+UP - moves to the top of the current method, or the previous method
- CTRL+ALT+DOWN - moves to the net method
- CTRL+ALT+HOME - first method in source
- CTRL+ALT+END - last method in source
- CTRL+ALT+MOUSE_WHEEL - scrolls through methods

Finding Classes

Allows you to find classes (using C# regular expressions). Use the**Search** ▸ **Find Classes** command to see a list of available classes that you can select. After you choose one, the IDE navigates to its declaration.

Finding Units

Depending on which language you are programming in, you can use a refactoring feature to locate namespaces or units. If you are using C#, you can use the Use the **Import Namespace** command to import namespaces into your code. If you are using the Delphi language, you can use the **Find Unit** command to locate and add units to your code file. For code that is written using the .NET framework, the Assembly Browser will open if the expression is not found. The Assembly Browser will allow you to browse for a type. The **Find Type** window allows regular expressions.

Code Templates

Code Templates allow you to have a dictionary of pre-written code, which can be inserted into your programs while you're working with the Code Editor. This reduces the amount of typing that you must do on a daily basis.

Use the links at the end of this topic to learn more about creating and using Code Templates.

Code Folding

Code folding lets you collapse sections of code to create a hierarchical view of your code and to make it easier to read and navigate. The collapsed code is not deleted, but hidden from view. To use code folding, click the plus and minus signs next to the code.

To-Do Lists

A **To-Do List** records tasks that need to be completed for a project. After you add a task to the **To-Do List**, you can edit the task, add it to your code as a comment, indicate that it has been completed, and then remove it from the list. You can filter the list to display only those tasks that interest you.

Keystroke Macros

You can record a series of keystrokes as a macro while editing code. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session. Recording a macro replaces the previously recorded macro.

Bookmarks

Bookmarks provide a convenient way to navigate long files. You can mark a location in your code with a bookmark and jump to that location from anywhere in the file. You can use up to ten bookmarks, numbered 0 through 9, within a file. When you set a bookmark, a book icon is displayed in the left gutter of the **Code Editor**.

Block Comments

You can comment a section of code by selecting the code in the **Code Editor** and pressing CTRL+/ (slash). Each line of the selected code is prefixed with `//` and will be ignored by the compiler. Pressing CTRL+/ will add or remove the slashes, based on whether the first line of the code is prefixed with `//`. When using the Visual Studio or Visual Basic key mappings, use CTRL+K+C to add and remove comment slashes.

Getting Started with Together

This section contains in introduction to modeling with **Borland Together**.

The two sample projects are designed to help you explore Together features while working with projects. Some of the special features include: UML modeling, patterns, generating project documentation.

In This Section

About Together
Provides a brief introduction to the feature set of Together. Use Together for building a UML model of your application.

About Together

Welcome to **Borland® Together®**, the award-winning, design-driven environment for modeling applications. Together includes features such as support for UML 2.0, OCL, patterns, Quality Assurance audits and metrics, source code refactoring and generation, IBM Rational Rose (MDL) format import, XMI format import and export, and automated documentation generation.

A key feature of Together, LiveSource™, keeps your Together diagrams synchronized with your source code in the Developer Studio 2006 Editor.

Together is an integral part of a complete ALM (Application Lifecycle Management) solution provided by Borland Software Corporation. This version of Together is a part of the new generation of the Borland's ALM solution named SDO (Software Delivery Optimization). SDO is Borland's vision and strategy for transforming software delivery to an incorporated and disciplined approach that aligns teams, technology and process to maximize the business value of software.

The Together features are tightly integrated with the Developer Studio 2006 environment. When Together support is activated, the following items are added or modified:

- **Diagram View**
- **Model View**
- **Object Inspector**
- **Tool Palette**

In addition, specific commands are added to the main menu and the context menus of the **Project Manager** and **Structure View**.

The following offer additional assistance, information, and resources:

- For information on how to use this Help system, see Help on Help
- Borland Together Home Page
- Borland Together Documentation
- Borland Product Support
- Borland Newsgroups

Not all features described in this Help system are available in all editions of the product.

Help on Help

This section includes information about the:

- Developer Studio 2006 Help
- Microsoft .NET Framework SDK Help
- Borland Developer Support Services and Web Sites
- Developer Studio 2006 *Quick Start* Guide
- Typographic Conventions Used in the Help

Developer Studio 2006 Help

The Developer Studio 2006 Help includes conceptual overviews, procedural how-to's, and reference information, allowing you to navigate from general to more specific information as needed.

Additionally, the persistent navigation panes in the Help window make it easier to locate and filter information. By default, no filter is set, allowing you to view all of the installed Help. However, to narrow the focus when searching the Help or using the index, use the **Filter by:** drop-down list on the **Content**, **Search**, and **Index** panes. To display the navigation panes, use the **View** ▸ **Navigation** menu command.

Tip: When navigating to a topic by using a link from another topic, the context of the topic you are viewing might not be obvious. To find the context of that topic within the **Content** pane, click the **Sync Contents** button on the toolbar of the **Borland Help** viewer.

Conceptual Overviews

The conceptual overviews provide information about product architecture, components, and tools that simplify development. If you are new to a particular area of development, such as modeling or ADO.NET, see the overview topic at the beginning of each section in the online Help.

At the end of most of the overviews, you will find links to related, more detailed information. Icons are used to indicate that a link leads to the .NET SDK, partner Help, or to a web site. The icons are explained later in this topic.

How-To Procedures

The how-to procedures provide step-by-step instructions. For development tasks that include several subtasks, there are *core procedures*, which include the subtasks required to accomplish a larger task. If you are beginning a development project and want to know what steps are involved, see the core procedure for the area you are working on.

In addition to the core procedures, there are several single-task procedures.

All of the procedures are located under **Procedures** in the **Content** pane of the Help window. Additionally, most of the conceptual overviews provide links to the pertinent procedures.

Reference Topics

The reference topics provides detailed information on subjects such as API elements, the Delphi language, and compiler directives.

All of the reference topics are located under **Reference** in the **Content** pane of the Help window. Additionally, most API references are underlined and link directly to the appropriate reference topic.

Context Sensitive F1 Help

Context sensitive Help is available throughout the IDE by selecting an item and pressing F1:

- In the **Code Editor**, select and highlight the entire element, such as a namespace, keyword, or method
- On a form **Design** tab, select the component
- In the **Messages** window, select a message
- Within IDE windows, such as the **Project Manager** or **Model View**, click within the window

Note: Pressing F1 on an element that is part of the VCL.NET framework displays the Developer Studio 2006 Help. Pressing F1 on an element that is part of the .NET framework displays the Microsoft .NET Help.

Microsoft SDK Help

Developer Studio 2006 is distributed with the both the Microsoft .NET Framework SDK and the Microsoft Platform SDK, which include extensive online Help. Where appropriate, the Developer Studio 2006 Help provides links to the SDK online Help. Alternatively, you can access the SDK Help directly from the **Content** pane of this Help system.

Borland Developer Support Services and Web Site

Borland offers a variety of support options to meet the needs of its diverse developer community. To find out about support, refer to www.borland.com/devsupport. From the web site, you can access many newsgroups where developers exchange information, tips, and techniques. The site also includes a list of books, technical documents, and Frequently Asked Questions (FAQ). Additionally, you can access the Borland Developer Network.

Developer Studio 2006 Quick Start Guide

The Developer Studio 2006*Quick Start* guide provides an overview of the Developer Studio 2006 development environment to help you install and start using the product right away. The *Quick Start* guide is shipped along with your product.

Typographic Conventions Used in the Help

The following typographic conventions are used throughout the Developer Studio 2006 online Help.

Typographic conventions

Convention	Used to indicate
`Monospace type`	Source code and text that you must type.
Boldface	Reserved language keywords or compiler options, references to dialog boxes and tools.
Italics	Developer Studio 2006 identifiers, such as variables or type names. Italicized text is also used for book titles and to emphasize new terms.
KEYCAPS	Keyboard keys, for example, the CTRL or ENTER key.
	A link to Web resources.
	An external link to Microsoft SDK documentation.
	An external link to documentation provided by Borland partners.

Managing the Development Life Cycle

The application development life cycle involves designing, developing, testing, debugging, and deploying applications. Developer Studio 2006 provides powerful tools to support this iterative process, including integrated source control, form design tools, the Delphi for .NET compiler, an integrated debugging environment, and installation and deployment tools.

In This Section

Managing the Development Cycle Overview
Provides a brief overview of the steps involved in managing the development cycle.

Using Source Control
Provides an overview of general source control concepts and specifics of the Developer Studio 2006 source control capabilities.

Designing User Interfaces
Provides an overview of designing user interfaces with the Developer Studio 2006 designers.

Together Features Overview

Compiling, Building, and Running Applications
Provides an overview of compiling, building, and running applications in the IDE.

Localizing Applications
Describes the Translation Tools available with Developer Studio 2006.

Overview of Debugging
Provides general debugging information and describes the debugging tools available in Developer Studio 2006.

Deploying Applications
Provides information about deploying applications.

Managing the Development Cycle Overview

The development cycle as described here is a subset of Application Lifecycle Management (ALM), dealing specifically with the part of the cycle that includes the implementation and control of actual development tasks. It does not include such things as modeling applications. Developer Studio 2006 provides a framework of tools that helps you manage and perform all of your development requirements.

These tools include:

- Requirements management
- Source control integration
- User interface design
- Code visualization capabilities
- Project building, compilation, and debugging capabilities

Requirements Management

Developer Studio 2006 provides full integration with CaliberRM requirements management software. Using this integration, you can add, remove, and update requirements for your software project within the Developer Studio 2006 IDE. This integration also enables you to create links between the requirement specification and the portions of the code within your software project that fulfill the requirement.

Source Control Integration

Developer Studio 2006 provides a full-featured direct integration with Borland StarTeam. This integration allows you to access your source control system in one of two ways:

- Manage project files within the source control system from the Developer Studio 2006 IDE.
- Invoke the source control system in a separate process.

Invoke the source control system in a separate process if you need to use specific features of that system, which are not exposed in the Developer Studio 2006 IDE. The source control application appears in a separate window.

In most cases, you manage your project files from within the Developer Studio 2006 IDE. The integration provided allows you to check-in, check-out, update, commit, and otherwise manage your source files using a simplified user interface. The integration supports the level of multi-user capabilities provided by your specific source control system.

User Interface Design

Developer Studio 2006 provides a rich environment for designing a .NET user interface. In addition to the Windows Form Designer, which includes a full set of visual components, the IDE gives you tools to build ASP.NET Web Forms, along with a set of Web Controls. Developer Studio 2006 also includes a VCL.NET Forms design tool, which allows you to build .NET applications using VCL components. The Designer offers a variety of alignment tools, font tools, and visual components for building many types of applications, including MDI and SDI applications, tabbed dialogs, and data aware applications.

Code Visualization

The Code Visualization feature of Developer Studio 2006 provides the means to document and debug your class designs using a visual paradigm. As you load your projects and code files, you can use the **Model View** to get both a hierarchical graphical view of all of the objects represented in your classes, as well as a UML-like model of your

application objects. This feature can help you visualize the relationships between objects in your application, and can assist you in developing and implementing.

Build, Compile, Run, and Debug

Developer Studio 2006 provides a full-featured build and compile system, along with an integrated debugger. The visual approach to building, compiling, and running your application makes the entire development process simpler than in the past. Projects with subprojects and multiple source files can be compiled all together, which is called *building*, or you can compile each project individually.

The integrated debugger allows you to set watches and breakpoints, and to step through, into, and over individual lines of code. A set of debugger windows provides details on variables, processes, and threads, and lets you drill down deeply into your code to find and fix errors.

Using Source Control

Borland's Developer Studio 2006 provides a full-featured direct integration with StarTeam, Borland's automated change and software configuration management (SCM) system. This integration lets you access StarTeam's rich feature set from within the IDE. The integration also provides some Developer Studio 2006-specific features to allow you to easily check in and check out Developer Studio 2006 project source files and manage your work more easily.

Source Control Basics

Each source control system consists of one or more centralized repositories and a number of clients. A repository is a database that contains not only the actual data files, but also the structure of each project you define.

Most source control systems adhere to a concept of a logical *project*, within which files are stored, usually in one or more tree directory structures. A source control system project might contain one or many Developer Studio 2006 projects in addition to other documents and artifacts. The system also enforces its own user authentication or, very often, takes advantage of the authentication provided by the underlying operating system. Doing so allows the source control system to maintain an audit trail or snapshot of updates to each file. These snapshots are typically referred to as *diffs*, for differences. By storing only the differences, the source control system can keep track of all changes with minimal storage requirements. When you want to see a complete copy of your file, the system performs a merge of the differences and presents you with a unified view. At the physical level, these differences are kept in separate files until you are ready to permanently merge your updates, at which time you can perform a *commit* action.

This approach allows you and other team members to work in parallel, simultaneously writing code for multiple shared projects, without the danger of an individual team member's code changes overwriting another's. Source control systems, in their most basic form, protect you from code conflicts and loss of early sources. Most source control systems give you the tools to manage code files with check-in and check-out capabilities, conflict reconciliation, and reporting capabilities. Most systems do not include logic conflict reconciliation or build management capabilities. For details about your particular source control system capabilities, refer to the appropriate product documentation provided by your source control system vendor.

Commonly, source control systems only allow you to compare and merge revisions for text-based files, such as source code files, HTML documents, and XML documents. The source control systems supported by Developer Studio 2006 allow you to include binary files, such as images or compiled code, in the projects you place under control. You cannot, however, compare or merge revisions of binary files. If you need to do more than store and retrieve specific revisions of of these types of files, you might consider creating a manual system for keeping tracking of the changes you make to binary files.

Repository Basics

Source control systems store copies of source files and difference files in some form of database repository. In some systems, such as CVS or VSS, the repository is a logical structure that consists of a set of flat files and control files. In other systems, the repositories are instances of a particular database management system (DBMS) such as InterBase, Microsoft Access, MS SQL Server, IBM DB2, or Oracle.

Repositories are typically stored on a remote server, which allows multiple users to connect, check files in and out, and perform other management tasks simultaneously. You need to make sure that you establish connectivity not only with the server, but also with the database instance. Check with your network, system, and database administrators to make sure your machine is equipped with the necessary drivers and connectivity software, in addition to the client-side source control software.

Some source control systems allow you to create a local repository in which you can maintain a snapshot of your projects. Over time the local image of your projects differs from the remote repository. You can establish a regular policy for merging and committing changes from your local repository to the remote repository.

Generally, it is not safe to give each member of your team a separate repository on a shared project. If you are each working on completely separate projects and you want to keep each project under source control locally, you can

use individual local repositories. You can also create these multiple repositories on a remote server, which provides centralized support, backup, and maintenance.

Working with Projects

Source control systems, like development environments, use the project concept to organize and track groups of related files. No matter which source control system you use, you create a project that maintains your file definitions and locations. You also create projects in Developer Studio 2006 to organize the various assemblies and source code files for any given application. Developer Studio 2006 stores the project parameters in a project file. You can store this file in your source control system project, in addition to the various code files you create. You might share your project file among all the developers on your team, or you might each maintain a separate project file. Most source control systems consider development environment project files to be binary, whether they are actually binary files or not. As a consequence, when you check a project file into a source control system repository, the source control system overwrites older versions of the file with the newer one without attempting to merge changes. The same is true when you pull a project, or check out the project file; the newer version of the project file overwrites the older version without merging.

Working with Files

The file is the lowest-level object that you can manage in a source control system. Any code you want to maintain under source control must be contained in a file. Most source control systems store files in a logical tree structure. Some systems, such as CVS, actually use terms like *branch*, to refer to a directory level. You can create files in a Developer Studio 2006 project and include them in your source control system, or you can pull existing files from the source control system. You can put an entire directory into the source control system, then you can check out individual files, multiple files, or entire subdirectory trees. Developer Studio 2006 gives you control over your files at two levels—at the project level within Developer Studio 2006 and in the source control system, through the Developer Studio 2006 interface to the source control system.

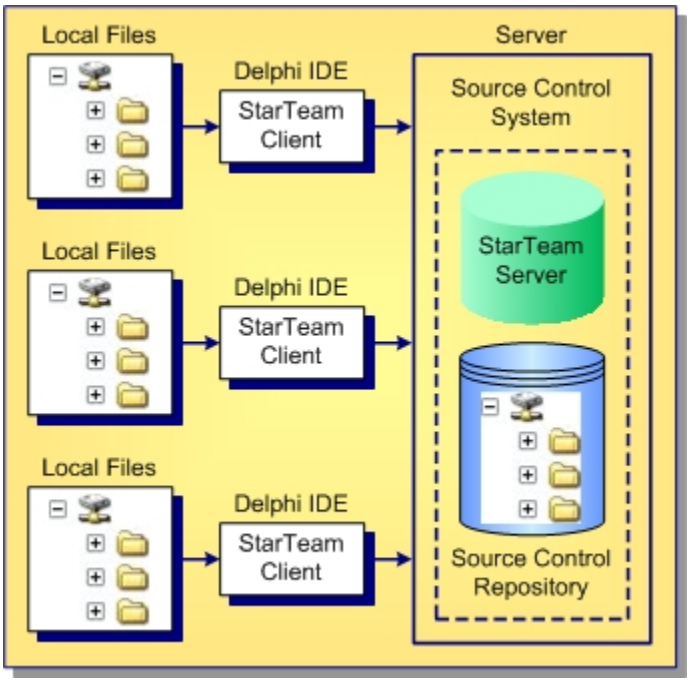
Note: The **History View** provides revision information for your local source files. The **History View** can be used to track changes you make to files as you work on them in the Designer or the **Code Editor**.

Using the StarTeam Integration

Borland's StarTeam integration for Developer Studio 2006 provides direct access to StarTeam features and functions from within the IDE. The StarTeam integration lets you use Developer Studio 2006 menus or embedded StarTeam Client elements to manage access to projects and files stored in the server repository, to maintain an audit trail of changes you make to the projects and files, and to resolve file revision conflicts.

The function and use of the StarTeam Client and the elements incorporated into Developer Studio 2006 are documented in detail in the *StarTeam User's Guide* and the *StarTeam Administrator's Guide*. StarTeam is a powerful tool, with comprehensive version control features and capabilities. We strongly recommend that you familiarize yourself with the StarTeam documentation before using this integration. All StarTeam documentation is available for download from the Borland web site at http://info.borland.com/techpubs/starteam/.

How Developer Studio 2006 Interacts with StarTeam



StarTeam consists of server and client components. On the server side, the StarTeam Server maintains a database repository that captures a complete snapshot of the source files in your project and incremental changes (deltas or differences) to those files. The StarTeam client is integrated seamlessly with Developer Studio 2006. You can place projects into and pull projects out of your source control repository, and check in, check out, merge, and compare files.

Note: The StarTeam integration for Developer Studio 2006 supports StarTeam 5.4, 6.0 and 2005 Servers.

StarTeam Client

The StarTeam integration for Developer Studio 2006 includes a StarTeam Client for the .NET Framework. You can launch the full StarTeam Client, or view the client as embedded elements of the IDE. These embedded StarTeam elements provide access to most of the commands and information available in the client's main window (also called the Project View Window).

The StarTeam client provided with the Delphi integration can only be started from within Delphi. There are no provisions for using StarTeam's command-line interface with the StarTeam integration for Delphi.

With the exception of the aforementioned items, the features supported by your StarTeam installation are supported by the Developer Studio 2006 integration. For example, if you have StarTeam Standard, which does not support tasks, requirements, or alternate property editors (APEs), the StarTeam integration for Developer Studio 2006 will not support tasks, requirements, or APEs. If you have StarTeam Enterprise, which supports tasks, your StarTeam integration will support tasks. If you have StarTeam Enterprise Advantage, your StarTeam integration will support tasks, requirements, and APEs. For more information about StarTeam, including a feature matrix, see the StarTeam product page on the Borland web site at http://www.borland.com/starteam/index.html.

Standard Version Control Support

The StarTeam integration provides support for standard version control operations. Using the integrated StarTeam Client, you can perform the following operations:

- Place and pull projects and project groups to and from a StarTeam repository
- Commit changes for the entire project
- Update the entire project with the latest revisions in the repository
- Check individual files in and out from the repository
- Add files to the StarTeam project
- Lock files for exclusive editing
- Compare two revisions of a file
- Revert files back to a prior revision

Developer Studio 2006 provides access to these operations through the **StarTeam** menu on the main menu bar, or through **StarTeam** context menus in the **Project Manager**.

Advanced Features

The integrated StarTeam Client lets you access advanced StarTeam features without leaving the development environment. Some of these include:

- Create and edit items other than files, such as change requests, requirements, tasks, and topics
- Apply labels to a file, an item, a group of files, or a group of items
- Establish process items and rules to help you link and track changes to your files

These features will help you assign and track responsibilities for tasks throughout your project. The client that can be launched from within Developer Studio 2006 provides even more features and functions for managing your files and projects, such as the ability to generate reports and charts, and administer user accounts and servers.

Developer Studio 2006 Features

Some features and behaviors of the StarTeam integration are specific to Developer Studio 2006. Beyond the embedded client, the most obvious of these is the support for Developer Studio 2006 projects and project groups. The StarTeam integration provides commands for placing, pulling, and updating Developer Studio 2006 projects and project groups, as well as for committing changes to all files in a project. Additionally, the integration provides quick access to StarTeam commands through context menus in the **Project Manager**.

The StarTeam integration works together with the Developer Studio 2006**History Manager** to display both local and StarTeam version information for the active file. You can use the **History Manager** to compare the contents of your current working file with revisions of the file in the StarTeam repository. You can also revert the contents of your working file to any StarTeam revision.

The StarTeam integration supports file renaming and deleting. When you rename or delete a file in your project, the StarTeam integration will automatically carry out the changes on the repository when you commit the project.

Similarly, if a team member has renamed or deleted files, and committed the changes, the changes are carried out in your local project when you update the project. This capability prevents loss of revision information when a file is renamed or moved.

Note: If the file renaming or deletions made in your local project conflict with changes made by another team member in the StarTeam Client, you must manually resolve the pending renaming or deletion of files. The **Pending Renames/Deletes** dialog box (**StarTeam ▸ Pending Renames/Deletes**) lets you commit any pending local file renames or deletions to the repository or cancel the pending operations.

When the **Structure View** displays the folder hierarchy for your StarTeam project, the **Structure View** includes a toolbar with the following features:

- A drop-down list of paths to the folders you've previously selected. Choose a path from the drop-down list to go to that StarTeam folder in the current hierarchy. The most recently selected folder sorts to the top.
- A Refresh button. Click this button to update the information in the current tree.
- A button for selecting which node in the folder hierarchy to show as the root folder, the project or the view. This button is available when the project and view are not at the same level.

Managing Requirements with CaliberRM

CaliberRM is a requirements management system that enables teams to fully define, manage and communicate changing requirements for software development projects. The integrated CaliberRM client provides direct access to features and functions from within theDeveloper Studio 2006 IDE. After you log in to CaliberRM through Developer Studio 2006, you can display and update data that is stored on the CaliberRM server. This data is accessible through both the integrated CaliberRM client and the Windows CaliberRM client. The Windows CaliberRM client is documented in detail in the *CaliberRM User Guide*. All CaliberRM documentation is available for download from the Borland web site at http://info.borland.com/techpubs/caliber_rm/.

The integration of CaliberRM into Developer Studio 2006 provides additional features not available from the standalone CaliberRM application.

- Direct logon to the CaliberRM server
- Linking between requirements and source code

Using the Integrated CaliberRM Client

CaliberRM consists of a client and a server component. An enhanced version of the client is available directly within the Developer Studio 2006 IDE. If you are logged on to the CaliberRM server, you can display the project requirements within the IDE. If you are not logged on, the logon screen displays.

Note: The CaliberRM integration for Developer Studio 2006 uses the CaliberRM 2005 Server, which is only available for Windows 2000 and Windows NT. See the *CaliberRM Installation Guide* for a complete list of system requirements and installation instructions for the server.

Logging On To the CaliberRM Server

You must log on to the CaliberRM server before you can display or update the requirements. You can log on to the server directly from the Developer Studio 2006 IDE. Once you are logged on, the integrated CaliberRM client displays in the bottom portion of the IDE.

Linking Between a Requirement and Source Code

You can select source code in your project and drag it into a requirement for your project. This creates a link that appears in the **Traceability** tab for that requirement. When you click the link, the project opens to that source code snippet.

Designing User Interfaces

A graphical user interface (GUI) consists of one or more windows that let users interact with your application. At designtime, those windows are called *forms*. Developer Studio 2006 provides a designer for creating Windows Forms, Web Forms, VCL Forms, and HTML pages. The Designer and forms help you create professional-looking user interfaces quickly and easily.

Using the Designer

When you create a Windows, Web, or Web Services application, the IDE automatically displays the appropriate type of form on the **Design** tab in the IDE. As you drop components, such as labels and text boxes, from the **Tool Palette** on to the form, Developer Studio 2006 generates the underlying code to support the application. You can use the **Object Inspector** to modify the properties of components and the form. The results of those changes appear automatically in the source code on the **Code** tab. Conversely, as you modify code with **Code Editor**, the changes you make are immediately reflected on the **Design** tab.

The **Tool Palette** provides dozens of controls to simplify the creation of Windows Forms, Web Forms, and HTML pages. When creating a Windows Form, for example, you can use the MainMenu component to create a customized main menu in minutes. After placing the component on a Windows Form, you type the main menu entries and commands in the boxes provided. The ContextMenu component provides similar functionality for creating context menus. There are also several dialog box components for commonly performed functions, such as opening and saving files, setting fonts, selecting colors, and printing. Using these components saves time and provides a consistent look and feel for the dialogs in your application.

As you design your user interface, you can undo and repeat previous changes to a form by choosing **Edit** ▸ **Undo** and **Edit** ▸ **Redo**. When you are satisfied with the appearance of the form, you can lock the components and form to prevent accidental changes by right-clicking the form and choosing **Lock Controls**.

Setting Designer Options

You can set options that effect the appearance and behavior of the Designer. For example, you can adjust the grid settings, and the style of generated code and HTML. To set these options, choose **Tools** ▸ **Options** and then use the **Windows Form Designer** and **HTML Option** dialog boxes.

Setting Designer Guidelines with VCL Components

You can use VCL or VCL.NET (with Delphi or C++) to setup components that are "aware" of their relation to other components on a form. For instance, when you drop a component on a form, it will leave a certain amount of space from the border of the form, depending on how the 'padding' property is set.

You can set properties to specify the distance between controls, shortcuts, focus labels, tab order, and maximum number of items (listboxes, menus).

The Code Developer can then use these components to create forms. when the Use Designer Guidelines option is enabled. If the Snap to Grid option is enabled, and Use Designer Guidelines is also enabled, the designer guidelines will take precedence.

See the Creating Designer Guidelines link at the end of this topic, to view the procedure for setting these guidelines.

Together Features Overview

This section provides an overview of the features provided by **Borland Together**.

In This Section

Modeling Overview
Describes what modeling with Together means in general.

Together Project Overview
Describes the Together projects.

Namespace and Package Overview
Describes Together namespaces and packages.

Together Diagram Overview
Describes the Together UML diagram.

Supported UML Specifications
Describes supported UML specifications.

Model Element Overview
Describes the model elements.

Annotation Overview
Describes the feature for annotating UML diagrams.

Shortcut Overview
Describes the shortcuts on UML diagrams.

Diagram Format Overview
Describes the UML diagram format.

Diagram Layout Overview
Describes the algorithms available to lay out UML diagrams.

Hyperlinking Overview
Describes the hyperlinking feature.

LiveSource Overview
Describes the LiveSource feature.

Transformation to Source Code Overview
Describes the transformation to source code feature.

OCL Support Overview
Describes support for Object Constraint Language.

Patterns Overview
Describes support for design patterns.

Refactoring Overview
Describes the concept of refactoring and introduces the refactoring operations included in Developer Studio 2006.

Quality Assurance Facilities Overview
Describes the Quality Assurance facilities.

Documentation Generation Facility Overview
Describes the documentation generation feature.

Import and Export Overview
Describes the import and export features.

Interoperability Overview
Describes the interoperability with other versions of Together.

Modeling Overview

Effective modeling with Together simplifies the development stage of your project. Smooth integration to Developer Studio 2006 provides developers with easy transition from models to source code.

The primary objective of modeling is to organize and visualize the structure and components of software intensive systems. Models visually represent requirements, subsystems, logical and physical elements, and structural and behavioral patterns.

While contemporary software practices stress the importance of developing models, Together extends the benefits inherent to modeling by fully synchronizing diagrams and source code.

Together Project Overview

Work in Together is done in the context of a project. A project is a logical structure that holds all resources required for your work. Together works with the following project types: design and implementation, and multiple project formats.

It is up to you to define which directories, archives, and files should be included in your project. You can set up project properties when the project is being created, and modify them further, using the **Object Inspector**.

Namespace and Package Overview

A namespace is an element in a model that contains a set of named elements that can be identified by name.

A project consists of one or more namespaces (or packages). A namespace and a package are almost synonyms: the term “namespace” is used for implementation projects, the term “package” is used for design projects.

A namespace (or a package) is like a box where you put diagrams and model elements. Contents of a namespace (package) can be displayed on a special type of the Class diagram.

Each project contains the default namespace (or package) just after its creation.

Together Diagram Overview

Diagrams can be thought of as graphs with vertices and edges that are arranged according to a certain algorithm.

Each diagram belongs to a certain diagram type (for example, UML 2.0 Class Diagram). A set of model elements available for use on a diagram depends on the diagram type.

Diagrams exist within the context of a namespace (or a package). You have to create or open a project or project group before creating a new diagram. When Together support is activated, the project-level package diagram is created by default. You can create the various UML diagrams in the project.

In addition to the standard properties of diagrams and their elements, you can create user properties, represented by the Name-Value pair.

Supported UML Specifications

The Object Management Group's Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.

Together supports UML to help you specify, visualize, and document models of your software systems, including their structure and design.

Refer to UML documentation for the detailed information about UML semantics and notation. The *UML (version): Superstructure* document defines the user level constructs required for UML. It is complemented by the *UML (version): Infrastructure* document which defines the foundational language constructs required for UML. The two complementary specifications constitute a complete specification for the UML modeling language.

UML 1.5 and UML 2.0

The set of available diagrams depends on your project type.

For design projects, both UML 1.5 and 2.0 are supported.

For implementation projects, UML 1.5 is only supported.

The version of UML is selected when a project is created. It cannot be changed later.

UML In Color

"UML In Color" is an optional profile to support the modeling in color methodology. Color modeling makes it possible to analyze a problem domain and easily spot certain classes during analysis. Together supports the use of the four main groups of the color-modeling stereotypes:

- Role
- MomentInterval, Mi-detail
- Party, Place, Thing
- Description

For each of these stereotypes you can choose a specific color to make your model more understandable at a glance. Note that the other stereotypes do not have associated colors.

See also *"Java Modeling in Color with UML: Enterprise Components and Process" by Coad, Lefebvre and De Luca*.

Model Element Overview

Model element is any component of your model that you can put on a diagram.

Model elements include nodes and links between them.

A set of available model elements depends on a current diagram type. Available model elements are displayed in the **Tool Palette**.

A link can have a label. You can move a label to any point of the link line.

Annotation Overview

The **Tool Palette** for UML diagram elements displays note and note link buttons for all UML diagrams. Use these elements to place annotation nodes and their links on the diagram.

Notes can be free floating or you can draw a note link to some other element to show that a note pertains specifically to it.

You can attach a note link to another link.

The text of notes linked to class diagram elements does not appear in the source code.

Shortcut Overview

A shortcut is a representation of an existing node element placed on the same or a different diagram.

Shortcuts facilitate reuse of elements, make it possible to display library classes on diagrams, and demonstrate relationships between the diagrams within the model.

You can create a shortcut to an element of any other project in the current project group. You can create a shortcut to an inner class or interface of another classifier. It is also possible to add a shortcut to an element from project References, including binary (`.dll`, `.exe`) files.

The small special symbol appears over a node to indicate a shortcut. It appears only if this node belongs to a different namespace or package.

Select a shortcut on your diagram and choose **Navigate To Element** on the context menu to navigate to the source element in the **Model View**.

Diagram Format Overview

Together stores diagrams in an XML-based format with the extension `.txv(diagram)` under the `ModelSupport_%PROJECTNAME% ModelSupport` folder of a project.

These files contain information about diagram elements such as layout, background color, stereotypes, and so on.

For example, the `(name).txvcls` file corresponds to a class diagram. All products that provide modeling (Borland Together for Visual Studio .NET, Borland Together ControlCenter, Borland Together Architect, Borland Together Edition for Eclipse, and Borland Together for JBuilder) support the same diagram format, which makes the diagrams compatible across the product line. You can copy and reuse diagrams created in the different products.

The diagram elements in fact belong to the parent default package (namespace) files (`default.txaPackage`). These files contain all the information about the elements and their properties, while the diagram files contain information about locations and dimensions of elements.

This version of Together uses the format with embedded model elements (created as filemates).

Diagram Layout Overview

You can customize diagram notation in several ways.

Together enables you to manage simple or complex diagrams with automated layout features that optimize the diagram layout for viewing or printing. Nodes and links on a diagram are arranged according to a certain algorithm.

It is also possible to adjust their arrangement manually.

There are several diagram layout algorithms available:

- **Autoselect**: several algorithms can be available for each diagram type. This option analyzes internal information of each algorithm, and selects the one that best suits the current diagram type. If autoselect: Each of the layout algorithms contains internal information about the types of diagrams it will work with and the numeric characteristics for the final quality of the produced layout when applied to each applicable diagram type. Several algorithms can be available for the same diagram type. The autoselect option uses such internal information and picks the best algorithm for the current diagram type.
- **Hierarchical**: this type of algorithm is most suitable to analyze hierarchical structure (for example study inheritance relationships). The Hierarchical algorithm originates from the Sugiyama algorithm. The algorithm draws the UML diagram hierarchically according to the preferences that you select.
- **Together**: algorithm applicable to all types of diagrams. It includes the layout options used in version 6.1 of Together ControlCenter and Together Edition for JBuilder.
- **Tree**: the algorithm draws a tree diagram in a tree layout. The algorithm draws the given graph in a tree layout according to its maximum spanning tree.
- **Orthogonal**: simple structural algorithm is used when hierarchy is not important. The Orthogonal algorithm uses heuristics to distribute diagram nodes among a lattice.
- **Spring Embedder**: Spring Embedder are force-directed layout algorithms that model the input graph as a system of forces and try to find a minimum energy configuration of this system. All edges are drawn as straight lines. This type of layout is especially suitable for projects with numerous diagram elements based on large amount of source code. When you lay out a graph according to the Spring Embedder layout algorithm, the program will simulate the graph as a physical model (masses and springs) and subject it to physical forces. The unnecessarily-long edges will be the most tense, and will try to contract the most. When the nodes and edges have been balanced, you will have a geometric representation of the graph.

Each algorithm has a set of specific options defined in the **Together** ▸ **(level)** ▸ **Diagram** ▸ **Layout** category of the **Options** dialog window.

Hyperlinking Overview

You can create hyperlinks from diagrams or diagram elements to other system artifacts and browse directly to them.

Why use hyperlinking?

Use hyperlinks for the following purposes:

- Link diagrams that are generalities or overviews to specifics and details.
- Create browse sequences leading through different but related views in a specific order; create hierarchical browse sequences.
- Link descendant classes to ancestors; browse hierarchies.
- Link diagrams or elements to standards or reference documents or generated documentation.
- Facilitate collaboration among team members.

Create a hyperlink from an existing diagram or one of its elements to any other diagram or diagram element in the project, or create a new diagram that will be hyperlinked to the current diagram.

You can also create hyperlinks from your diagrams to external documents such as files or URLs. For most users, such hyperlinking will probably take the form of documents on a LAN or document server or URLs on the company intranet. However, you can also easily link to online information such as newsgroups or discussion forums. If it is available online, you can link to it.

Hyperlink types

You can create hyperlinks to:

- An existing diagram or diagram element anywhere in the project group
- A new diagram (it will be created on-the-fly)
- A document or file on a local or remote storage device
- A URL on your company intranet or the Internet

LiveSource Overview

LiveSource™ is the key feature of Together that keeps your model and source code in sync. That is why it applies to implementation projects only.

When a Class diagram is created in an implementation project, it is immediately synchronized with the implementation code. When you change a Class diagram, Together updates the corresponding source code.

Together allows you to synchronize different aspects of your project in several ways.

Use the Reload command to refresh the Together model from the source code.

About MDA

Together supports the OMG's Model Driven Architecture (MDA) initiative.

MDA is an evolving conceptual architecture for a set of industry-wide technology specifications that will support a model-driven approach to software development.

MDA is supported by UML, XMI, and other technologies.

Doc comment properties

Some properties that are defined for the model elements and members in the **Object Inspector**, are presented in the source code as language-specific doc comments. In particular, these properties are: author, since, version, stereotype, associates, and so on. When such comments are encountered in the source code, they are reverse engineered to model properties.

Doc comments are presented as XML tags, preceeded by `///` (for C# projects).

So doing, if the properties of an element are presented in the legacy format and one of these properties is changed to the new format `'<property> value</property>`, all the other properties are also converted.

Transformation to Source Code Overview

Together enables you to generate source code based on a language-neutral design project.

About transformation to source code

You can generate source code from the class diagrams of your design project and add this source code to a project in one of the supported languages. The target implementation project must already exist in the same project group.

Alternatively, you can import source code from an external design project into your current implementation project.

Name mapping

You can force Together to generate different names for your model elements in the source code. For example, you can have `ClassItem` in your source code for the `Class1` element in your model.

This feature is especially useful, if your model names are not English. You can use names in Japanese and other languages on your diagrams, but keep names in Latin alphabet in your code.

If you enable this feature, the file `codegen_map.xml` is created in the model support folder of the source design project. You can edit it with any XML or text editor. This file contains a mapping table, where each entry (model element) has two names: one for the source design project (attribute `name`), and another one for the destination implementation project (attribute `alias`). There are several sections in this file: Class, Attribute, Operation and Package for UML 1.5 projects, and Class and Package for UML 2.0 projects. Attributes name must be unique for all entries in a section.

You can optionally create an XML file with the same name and structure in a folder of any package.

Then, if you transform your project to source code and the name mapping feature is enabled, Together searches for the `codegen_map.xml` file for each model element. If the file is absent for a current package, Together searches in a parent package, and so on.

Note: If you add a new element to your model later and then transform the project to source code, Together adds a new entry for this item to the corresponding `codegen_map.xml` file. The existing entries are not changed.

OCL Support Overview

About OCL

The Object Constraint Language (OCL) is a textual language, especially designed for use in the context of diagrammatic languages such as the UML. OCL was added to UML, as it turned out a visual diagram-based language is limited in its expressiveness.

OCL 2.0 is the newest version of the OMG's constraint language to accompany their suit of Object Oriented modelling languages.

The use of OCL as an accompanying constraint and query language for modelling with these languages is essential.

Note: Portions of this product include the Object Constraint Language Library, courtesy of Kent University, United Kingdom. See http://www.cs.kent.ac.uk/projects/ocl/

OCL constraint and expression

OCL constraint

The **Tool Palette** on some types of diagrams (for example, UML 2.0 Class Diagram) contains buttons that enable you to create OCL constraints as design elements on diagrams, and link these constraints with the desired context.

You can show or hide constraint elements for the better presentation of your diagrams.

OCL support for constraints provides error highlighting. The text of the constraint is validated when the constraint is linked to its context. The valid constraints are displayed in the regular font; invalid constraints, or OCL expressions with syntax errors, are displayed in a red font.

Constrained elements are marked with the decorators. The decorators are small icons attached to the context elements of constraints. If a constraint is valid the decorator is green; otherwise the decorator is red. If the constraints are concealed, you can still monitor the validity of constraints by means of the decorators.

Any OCL constraint contains an OCL expression.

OCL expression

For OCL expressions without object constraints (expressions as properties of other nodes), no validation is performed since no valid OCL context can be set for these elements.

Supported diagram types

OCL is supported for the following diagram types:

Diagram types with OCL support

Diagram type	Version of UML	How support is provided
Class (class, namespace, package)	UML 1.5, 2.0	Creating object constraints is supported.
Interaction (Sequence and Communication)	UML 2.0	State invariant constraints for lifelines and constraints for the operands of the combined fragments as OCL expressions.
State Machine	UML 2.0	Guard conditions of transitions as OCL expressions.

Use Case	UML 2.0	Pre- and post-condition constraints for the behavior associated with the use cases as OCL expressions. For example, an interaction chosen as a behavior.

Patterns Overview

Patterns provide software developers with powerful reuse facilities. Rather than trying to tackle each design problem from the very outset, you can use the predefined patterns supplied with Together. The hierarchy of patterns is defined in the **Pattern Registry**. You can manage and logically arrange your patterns using the **Pattern Organizer**.

Patterns are intended to:

- Create frequently used elements
- Modify existing elements
- Implement useful source code constructions or project groups in your model

Pattern Registry

The **Pattern Registry** defines the virtual hierarchy of patterns. You can create virtual folders and group the patterns logically to meet your specific requirements. All operations with the contents of the **Pattern Registry** are performed in the **Pattern Organizer** and synchronized with the **Pattern Registry**.

Pattern Organizer

The **Pattern Organizer** enables you to logically organize patterns (using virtual trees, folders and shortcuts), and view and edit the pattern properties. You will be working with shortcuts, not with the actual patterns. Because of this, shortcuts to the same pattern may be included in several folders.

Code templates

Together supports code templates as a way to provide backward compatibility with the legacy Together ControlCenter projects. You can copy the folders with your legacy source code templates to the `Patterns` subfolder of your Together installation directory, and use these templates to create elements in implementation projects.

Code templates are text files with the language-specific extensions that use macros to be substituted with real values when the templates are applied. Therefore, code templates can be regarded as forms ready for "filling in" for a specific instance. A code template consists of a template file containing source code, and a properties file that contains macro descriptions and their default values.

Code templates are stored in the `Patterns\templates` directory of your Together installation using the following structure:

```
/<language>/<category>/<template name>
```

where <category> is CLASS, LINK or MEMBER. Each <template_name> folder contains the following files:

- %Name%.<ext>
- <template_name>,properties (optional)

Design patterns

A design pattern is an XML file that contains a sequence of statements or actions, required to create entities and links and set their properties. Each statement creates either one model element or one link between the model elements.

In addition to creating new elements, you can use design patterns to add members to a container element. The pattern that you are applying to the specified container element should have its Use Existent property set as True.

You can then apply the pattern to the container element you want to modify. For example, if you want to add several methods stored in a class as pattern to an existing class, then you have to apply that pattern to the diagram where that class exists.

The design patterns are stored as XML files in the Patterns directory of your Together installation.

Patterns as First Class Citizens

A First Class Citizen (FCC) pattern is a specific type of design pattern that contains information about the pattern name and the role of each participant. When applied to a diagram, FCC patterns create their own entities and display on the diagram with links to the created entities. Such patterns enable further modification by means of adding new participants.

Patterns as First Class Citizens are represented by GoF patterns.

A pattern is displayed on a diagram as an oval with the pattern name and an expandable list of participants. Each participant is connected with the pattern oval by a link, labeled with the participant's role.

FCC patterns generate source code, but the oval FCC pattern elements do not. The entities created by patterns are stored in the diagram files.

Stub implementation pattern

When you create an inheritance link between a class and another abstract class or interface, the methods and members are not automatically added to the child class. This problem is solved using the Stub implementation pattern. You can also create an implementation link and stub implementation in one step by using the Implementation link and stub pattern.

If the destination of a link is an interface, the pattern makes the class-source implement that interface, and creates in a class the stubs for all of the methods found in the interface and all of its parent interfaces.

If the destination link is an abstract class, this pattern makes the class-source extend the class-destination, and makes stubs for all of the constructors found in the class-destination. These constructor stubs call the corresponding constructors in the class-destination.

You can find the Implementation link and stub pattern in the Pattern Wizard by clicking the Link by Pattern or Node by Pattern buttons in the **Tool Palette**, or by using the Create by Pattern context menu for a class.

The Implementation link and stub pattern creates the following members of interfaces and abstract classes:

- Methods
- Functions
- Subroutines
- Properties
- Indexers
- Events

Refactoring Overview

Together provides extensive support for refactoring your implementation projects.

Refactoring means rewriting existing source code with the intent of improving its design rather than changing its external behavior. The focus of refactoring is on the structure of the source code, changing the design to make the code easier to understand, maintain, and modify.

The refactoring features provided by Together affect both source code and model. As a result, your project is consisting after refactoring, even if it includes UML diagrams.

The primary resource book on refactoring is *Refactoring - Improving the Design of Existing Code by Martin Fowler (Addison - Wesley, 1999)*.

Quality Assurance Facilities Overview

Together provides audits and metrics as Quality Assurance features to unobtrusively help you enforce company standards and conventions, capture real metrics, and improve what you do. Although audits and metrics are similar in that they both analyze your code, they serve different purposes.

Audits and metrics are run as separate processes. Because the results of these two processes are different in nature, Together provides different features for interpreting and organizing the results. Note that some of the features and procedures described in this section apply to both audits and metrics while some are specific to one or the other.

Audits

When you run audits, you select specific rules to which your source code should conform. The results display only the violations of those rules so that you can examine each problem and decide whether to correct the source code. Together provides a wide variety of audits to choose from, ranging from design issues to naming conventions, along with descriptions of what each audit looks for and how to fix violations. You can create, save, and reuse sets of audits to run. Together ships with a predefined saved audit set `(current.adt)` and you can create your own custom sets of audits to use.

Warning: This feature is available for implementation projects only.

Metrics

Metrics evaluate object model complexity and quantify your code. It is up to you to examine the results and decide whether they are acceptable. Metrics results can highlight parts of code that need to be redesigned, or they can be used for creating reports and for comparing the overall impact of changes in a project.

Together supports a wide range of metrics. See the descriptions of available metrics in the Metrics dialog window.

You can define, save, and reuse sets of metrics.

Along with the full set of metrics, Together provides tips for using metrics and interpreting results.

Warning: This feature is available for implementation projects only.

Bar chart

Metrics results can also be viewed graphically. Two graphic views allow you to summarize metrics results: bar charts and Kiviat charts. Both charts are invoked from the context menu of the table. Use the Kiviat chart for rows and the bar chart for columns.

The bar chart displays the results of a selected metric for all packages, classes, and/or operations.

The bar color reflects conformance to the limiting values of the metric in reference:

- Green represents values that fall within the permissible range.
- Red represents values that exceed the upper limit.
- Blue represents values that are lower than the minimal permissible value.
- A thin vertical red line represents the upper limit and a thin vertical blue line represents the lower limit.

Kiviat chart

Use the Kiviat chart for rows and the bar chart for columns.

The Kiviat chart demonstrates the analysis results of the currently selected class or package for all the metrics that have predefined limiting values. The metrics results are arranged along the axes that originate from the center of the graph.

Each axis has a logarithmic scale with the logarithmic base being the axis metric upper limit so that all upper limit values are equidistant from the center. In this way, limits and values are displayed using the following notation:

- Upper limits are represented by a red circle. Any points outside the red circle violate the upper limit.
- Lower limits are represented by blue shading, showing that any points inside the blue area violate the lower limit. Note that blue shading does not show up in areas of the graph with lower limits of 1 or 0.

As the mouse cursor hovers over the chart, the Visual Studio status bar displays information about the metrics or metrics values that correspond to the tick marks.

- The actual metrics show up in the form of a star with metric values drawn as points.
- Green points represent acceptable values.
- Blue points represent values below the lower limit.
- Red points represent values exceeding the upper limit.
- Scale marks are displayed as clockwise directional ticks perpendicular to the Kiviat ray.
- Lower limit labels are displayed as counterclockwise directional blue ticks perpendicular to the Kiviat ray.

Sets of audits and metrics

Both Audits and Metrics dialog boxes display the set of all available audits and metrics. When you open a project, a default subset is active. Active audits and metrics are indicated by checkmarks. If you open the desired dialog and click Start, all of the active audits/metrics are processed.

You will not want to run every audit or metric in the default active set every time, but rather some specific subset. Together enables you to create saved sets of active audits and metrics that can be loaded and processed as you choose. To do that, use the Load Set and Save Set buttons on the toolbar of the Audits and Metrics dialog windows. You can always restore the default active set using the Set Defaults button in the Audits dialog. Refer to the Audits dialog for description of controls.

Use the default active audits set or any saved set as the basis for creating a new saved set. By default, audit sets are saved in the QA folder under the Together installation.

Documentation Generation Facility Overview

This feature automatically generates documentation for your project. Use this feature to illustrate you programme with the documentation in the HTML format. You can update this automatically generated documentation when your project changes, or edit this documentation manually afterwards.

Documentation files

All the documentation that Together generates is written to a single directory that you specify in the `Output` folder of the **Generate HTML** dialog box. By default, the generated documentation opens in your external web browser. The browser opens with a frameset to display the generated documentation. If you choose not to open the documentation immediately, you can open it later using the `index.html` file found on the root of the documentation directory specified in the **Generate HTML** dialog box.

HTML documentation frames

The HTML documentation contains three frames:

- **Diagram frame**, when **Include diagrams** option is turned on
- **Project** and **Overview frame**, when **Include navigation tree** option is turned on
- **PackageList** and **PackageOverview frame**, when **Include navigation tree** option is turned off
- **Documentation frame**

You can click the **Project** tab in the lower left frame and expand the nodes in the project tree view. Notice that clicking a class name in the Project tab opens the documentation in the lower right pane (the **Documentation frame**). When you select a diagram in the **Project** tab, it opens in the **Diagram frame**. Elements in the **Diagram** frame are hyperlinked to the **Documentation** frame. If you select an element in the **Diagram** frame, its contents are displayed in the **Documentation** frame.

The **Documentation** frame displays the documentation of your source code and diagrams, and includes everything you would expect when generating HTML documentation. The top of the **Documentation** frame contains a navigation bar for browsing your project documentation.

The **Project** tab contains a tree representation of the project. Expand the nodes to reveal individual diagrams and elements. Clicking a class or interface opens the related documentation in the **Documentation** frame.

Import and Export Overview

You can share model information with other systems by importing and exporting model information, or by sharing project files:

Import and export features

Feature	Description
Exporting diagrams to images	You can save diagrams in several formats, including: Bitmap image (`BMP`) Enhanced windows metafile (`EMF`) Graphics interchange (`GIF`) JPEG file interchange (`JPG`) W3C portable network graphics (`PNG`) Tag image file (`TIFF`) Windows metafile (`WMF`)
Importing IBM Rational Rose (MDL) models	It is possible to convert models designed in IBM Rational Rose 2003 to the format of Together. The following file formats are supported: `.mdl, .ptl, .cat`, and `.sub`. For import, you create a new design UML 1.5 project based on the IBM Rational Rose project.
Importing from XMI Exporting to XMI	XMI (XML Metadata Interchange) enables the exchange of metadata information. Using XMI, you can exchange models across languages and applications. For example, if you have a modeling project created with a tool other than Together, you can import it to Together as an XMI file for extension or as the basis of a new project. Likewise, you can export Together projects for use in other applications. The result in each case is a single, portable .xml file. Together supports UML 1.3 Unisys XMI interchange for 8 types of UML diagrams. This feature is available for design projects that comply with the UML 1.5 specification.
Importing from other versions of Together	See Interoperability Overview
Export a Quality Assurance metric chart to image	Create a chart and then export it to image.

Interoperability Overview

This version of Together supports compatibility with other versions. This compatibility is based on the common diagram format, which enables you to reuse models created in the different editions of Together:

- Borland Together ControlCenter (TCC)
- Borland Together Architect (TAR)
- Borland Together for Microsoft Visual Studio .NET (TVS)
- Borland Together for Eclipse (TEC)
- Borland Together for JBuilder (TJB)
- Borland Together Designer 2005 and Borland Together Developer 2005, for PrimeTime (TPT)

Compiling, Building, and Running Applications

As you develop your application, you can compile, build, and run the application in the IDE. While all three operations can produce either an executable (.exe) or an assembly (.dll), they differ slightly in behavior:

- Compiling a project compiles the files in the current project that have changed since the last build and any files that depend on them. It does not execute the application.
- Building a project compiles all of the source code in the current project, regardless of whether any source code has changed. Building is useful when you are unsure which files have changed, or if you have changed project or compiler options.
- Running a project compiles any changed source code and, if the compile is successful, executes your application, allowing you to use and test it in the IDE.

Use the commands on the **Project** and **Run** menus to compile, build, and run your project.

Compiler Options

You can set many of the compiler options for a project by choosing **Project** ▸ **Options** and selecting the **Compiler** page. Most of the options on the **Compiler** page correspond to a compiler option and are described in the online Help for that page.

For Visual Basic and C# projects, you can save compiler options as an *option set*. This lets you quickly change options based on your development activity. For example, you can set compiler options specific to debugging your project, and then change the option set when you are done debugging it.

If you need to specify additional compiler options, you can invoke the compiler from the command line. For a complete list of the Delphi compiler options and information about running the Delphi compiler from the command line, see **Delphi Language Guide** in the **Content** pane. For a complete list of the C# compiler options and information about running the C# compiler from the command line, see the .NET Framework SDK online Help.

As you compile your project, you can display the current compiler options in the **Messages** window. Choose **Tools** ▸ **Options** ▸ **Environment Options** and select the **Show command line** option. The next time you compile a project, the command used to compile the project and the response file will displayed in the **Messages** window. The response file lists the compiler options and the files to be compiled.

Compiler Status and Information

You can display compiler information in the IDE during and after compilation. You can request that a status dialog be displayed each time you compile a project by choosing **Tools** ▸ **Option** ▸ **Environment Options** and checking the **Show Compiler Progress** check box.

After you compile a project, you can display information about it by choosing **Project** ▸ **Information**. The resulting **Information** dialog box displays the number of lines of source code compiled, the byte size of your code and data, the stack and file sizes, and the compile status of the project.

Compiler Errors

As you compile a project, compiler messages are displayed in the **Messages** window. For an explanation of a message, select the message and press F1.

Refactoring Applications

Refactoring is a technique you can use to restructure and modify your code in such a way that the intended behavior of your code stays the same. Developer Studio 2006 provides a number of refactoring features that allow you to streamline, simplify, and improve both performance and readability of your application code.

In This Section

Refactoring Overview
Describes the concept of refactoring and introduces the refactoring operations included in Developer Studio 2006.

Symbol Rename Overview (Delphi, C#, C++)
Describes the rename feature.

Refactoring Code
Describes how to use the refactoring features in Developer Studio 2006.

Previewing and Applying Refactoring Operations
Describes how to preview and apply refactoring operations.

Sync Edit Mode (Delphi, C#, C++)
Describes Sync Edit Mode.

Extract Method Overview (Delphi)
Describes the Extract Method refactoring.

Find References Overview (Delphi, C#, C++)
Describes the Find References refactoring feature.

Declare Variable and Declare Field Overview (Delphi)
Describes the concepts of declaring variables and fields through refactoring.

Extract Resource String (Delphi)
Describes the refactoring feature Extract Resource String.

Finding References
Describes how to use the Find References features.

Undoing a Refactoring (Delphi, C#)
Describes the Undo refactoring operation.

Finding Units and Using Namespaces (Delphi, C#)
Describes the refactoring feature that allows you to locate namespaces or units.

Refactoring Overview

Refactoring is a technique you can use to restructure and modify your existing code in such a way that the intended behavior of your code stays the same. Refactoring allows you to streamline, simplify, and improve both performance and readability of your application code.

Each refactoring operation acts upon one specific type of identifier. By performing a number of successive refactorings, you build up a large transformation of the code structure, and yet, because each refactoring is limited to a single type of object or operation, the margin of error is small. You can always back out of a particular refactoring, if you find that it gives you an unexpected result. Each refactoring operation has its own set of constraints. For example, you cannot rename symbols that are imported by the compiler. These are described in each of the specific refactoring topics.

Developer Studio 2006 includes a refactoring engine that evaluates and executes the refactoring operation. The engine also displays a preview of what changes will occur in a refactoring pane that appears at the bottom of the **Code Editor**. The potential refactoring operations are displayed as tree nodes, which can be expanded to show additional items that might be affected by the refactoring, if they exist. Warnings and errors also appear in this pane. You can access the refactoring tools from the **Main** menu and from context-sensitive drop down menus.

Developer Studio 2006 provides the following refactoring operations:

- Symbol Rename (Delphi, C#, C++)
- Extract Method (Delphi)
- Declare Variable and Field (Delphi)
- Sync Edit Mode (Delphi, C#)
- Find References (Delphi, C#, C++)
- Extract Resourcestring (Delphi)
- Find Unit (Delphi)
- Use Namespace (C#)
- Undo (Delphi, C#)
- Change Parameters (Delphi)

Symbol Rename Overview (Delphi, C#, C++)

Renames identifiers and all references to the target identifier. You can rename an identifier if the original declaration identifier is in your project or in a project your project depends on, in the Project Group. You can also rename an identifier if it is an error identifier, for instance, an undeclared identifier or type.

The refactoring engine enforces a few renaming rules:

- You cannot rename an identifier to a keyword.
- You cannot rename an identifier to the same identifier name unless its case differs.
- You cannot rename an identifier from within a dependent project when the project where the original declaration identifier resides is not open.
- You cannot rename symbols imported by the compiler.
- You cannot rename an overridden method when the base method is declared in a class that is not in your project.
- If an error results from a refactoring, the engine cannot apply the change. For example, you cannot rename an identifier to a name that already exists in the same declaration scope. If you still want to rename your identifier, you need to rename the identifier that already has the target name first, then refresh the refactoring. You can also redo the refactoring and select a new name. The refactoring engine traverses parent scopes, searching for an identifier with the same name. If the engine finds an identifier with the same name, it issues a warning.

Rename Method

Renaming a method, type, and other objects is functionally the same as renaming an identifier. If you select a procedure name in the **Code Editor**, you can rename it. If the procedure is overloaded, the refactoring engine renames only the overloaded procedure and only calls to the overloaded procedure. An example of this rule follows:

```
procedure Foo; overload;
procedure Foo(A:Integer); overload;
Foo();
Foo;
Foo(5);
```

If you rename the first procedure *Foo* in the preceding code block, the engine renames the first, third, and fourth items.

If you rename an overridden identifier, the engine renames all of the base declarations and descendent declarations, which means the original virtual identifier and all overridden symbols that exist. An example of this rule follows:

```
TFoo = class
    procedure Foo; virtual;
end;

TFoo2 = class(TFoo)
    procedure Foo; override;
end;

TFoo3 = class(TFoo)
    procedure Foo; override;
end;

TFoo4 = class(TFoo3)
    procedure Foo; override;
end;
```

Performing a rename operation on *Foo* renames all instances of *Foo* shown in the preceding code sample.

Extract Method Overview (Delphi)

Use the Extract Method refactoring operation to change a code fragment into a method whose name describes the purpose of the method. The Extract Method feature analyzes any highlighted code. If that code is not extractable to a method, the refactoring engine warns you. If the method can be refactored, the refactoring engine creates a new method outside of the current method. The refactoring engine then determines any parameters, generates local variables, determines the return type, and prompts the user for a new name. The refactoring engine inserts a method call to the new method in the location of the old method.

There are certain limitations to the extract method refactoring. They include:

- Cannot extract expressions, only statements.
- Cannot extract statements that include a call to inherited in Delphi.
- Cannot extract statements that are contained within a with statement.
- Cannot extract statements that call a local procedure or function.

If you select an expression and choose the Extract Method command, your selection will be expanded to include the entire statement. If the expression in your statement is used as a result, the extracted code returns a function result in place of the expression.

Extract Resource String (Delphi)

Extracting resource strings helps centralize string definitions which can then be more easily translated, if necessary. You can extract string values to resource strings that are defined in the **resourcestring** section of your code file. If there is no **resourcestring** section in your code, the refactoring engine creates one following either the implementation keyword or the uses list.

You cannot create a resource string from the following elements:

- **Constants.** For example, `const A = 'abcdefg';` cannot be extracted to a resource string.
- **Constants in Parameters.** For example, in `MyProc(A, B:Integer; C: string='test');` the string cannot be extracted to a resource string.
- **Resource Strings.** For example, `resourcestring A = 'test';` is already a resource string.

Declare Variable and Declare Field Overview (Delphi)

You can use the Refactoring feature to create variables and fields. This feature allows you to create and declare variables and fields while coding without planning ahead. This topic includes information about:

- Declare Variable
- Initial Type Suggestion
- Declare Field

Declare Variable

You can create a variable when you have an undeclared identifier that exists within a procedure block scope. This feature gives you the capability to select an undeclared identifier and create a new variable declaration with a simple menu selection or keyboard shortcut. When you invoke the **Declare Variable** dialog, the dialog contains a suggested name for the variable, based on the selection itself. If you choose to name the variable something else, the operation succeeds in creating the variable, however, the undeclared identifier symbol (Error Insight underlining) remains.

Variable names must conform to the language rules for an identifier. In Delphi, the variable name:

- Cannot be a keyword.
- Cannot contain a space.
- Cannot be the same as a reserved word, such as if or begin.
- Must begin with a Unicode alphabetic character or an underscore, but can contain Unicode alphanumeric characters or underscores in the body of the variable name.
- In the Delphi language, the type name can also be the keyword string.

Note: The .NET SDK recommends against using leading underscores in identifiers, as this pattern is reserved for system use.

Note: On the dialog that appears when you choose to declare a variable, you can set or decline to set an initial value for the variable.

Initial Type Suggestion

The refactoring engine attempts to suggest a type for the variable that it is to create. The engine evaluates binary operations of the selected statement and uses the type of the sum of the child operands as the type for the new variable. For example, consider the following statement:

```
myVar := x + 1;
```

The refactoring engine automatically assumes the new variable *myVar* should be set to type Integer, provided *x* is an Integer.

Often, the refactoring engine can infer the type by evaluating a statement. For instance, the statement `If foo Then...` implies that *foo* is a Boolean. In the example `If (foo = 5) Then...` the expression result is a Boolean. Nonetheless, the expression is a comparison of an ordinal (*5*) and an unknown type (*foo*). The binary operation indicates that *foo* must be an ordinal.

Declare Field

You can declare a field when you have an undeclared identifier that exists within a class scope. Like the Declare Variable feature, you can refactor a field you create in code and the refactoring engine will create the field declaration

for you in the correct location. To perform this operation successfully, the field must exist within the scope of its parent class. This can be accomplished either by coding the field within the class itself, or by prefixing the field name with the object name, which provides the context for the field.

The rules for declaring a field are the same as those for declaring a variable:

- Cannot be a keyword.
- Cannot contain a space.
- Cannot be the same as a reserved word, such as if or begin.
- Must begin with a Unicode alphabetic character or an underscore, but can contain Unicode alphanumeric characters or underscores in the body of the field name.
- In the Delphi language, the type name can also be the keyword string.

Note: Leading underscores on identifiers are reserved in .NET for system use.

You can select a visibility for the field. When you select a visibility that is not private or strict private, the refactoring engine performs the following operations:

- Searches to find all child classes.
- Searches each child class to find the field name.
- Displays a red error item if the field name conflicts with a field in a descendant class.
- You cannot apply the refactoring if it conflicts with an existing item name.

Sample Refactorings

The following examples show what will happen when declaring variables and fields using the refactoring feature.

Consider the following code:

```
TFoo = class
private
  procedure Foo1;
end;
...

implementation

procedure TFoo.Foo1;
begin
  FTestString := 'test';    // refactor TestString, assign field
end;
```

Assume you apply a Declare Field refactoring. This would be the result:

```
TFoo = class
private
  FTestString: string;
  procedure Foo1;
end;
```

If you apply a Declare Variable refactoring instead, the result is:

```
procedure TFoo.Foo1;
var                             // added by refactor
  TestString: string;    // added by refactor
begin
  TestString := 'test';     // added by refactor
  TestString := 'whatever';
end;
```

Find References Overview (Delphi, C#, C++)

Sometimes, you may not want to change code, but want to find references to a particular identifier. The refactoring engine provides **Find References**, **Find Local References**, and **Find Declaration Symbol** commands.

Both **Find References** and **Find Local References** commands provide you with a hierarchical list in a separate **Find References** window, showing you all occurrences of a selected reference. If you choose the **Find References** command, you are presented with a treeview of all references to your selection in the entire project. If you want to see local references only, meaning those in the active code file, you can select the **Find Local References** command from the **Search** menu. If you want to find the original declaration within the active Delphi code file, you can use the **Find Declaration Symbol** command. The **Find Declaration Symbol** command is only valid in Delphi and does not apply to C#.

Sample Refactoring

The following sample illustrates how the Find References refactoring will proceed:

```
1 TFoo = class
2   loc_a: Integer;            // Find references on loc_a finds only
3   procedure Foo1;            // this line (Line 2) and the usage
4 end;                         // in TFoo.Foo1 (Line 15)

5 var
6  loc_a: string;            // Find references on loc_a here
                             // finds only this line (Line 6) and
                             // the usage in procedure Foo (Line11)
7 implementation

8 {$R *.nfm}

9 procedure Foo;
10 begin
11  loc_a := 'test';
12 end;

13 procedure TFoo.Foo1;
14 begin   //
15   loc_a:=1;
16 end;
```

Change Parameters Overview (Delphi)

Adding or removing a parameter from a function is a commonly performed and tedious programming task. Developer Studio 2006 provides the **Change Parameters** refactoring to automate this task. You can use **Change Parameters** to add, remove, and rearrange function parameters.

To use this refactoring, select a function name in the **Code Editor** and choose **Refactor** ▸ **Change Params**.

When you use the **Change Parameters** refactoring, the following function signature conflicts can occur:

- A descendant class contains an override for the function you are refactoring. When you refactor the function, any functions that override the refactored function will also be refactored.
- A descendent class contains an overloaded version of the function that has the same signature as the refactored version. When you refactor the function, the overload is changed to an override.
- A descendent class has an overridden method that matches the original signature. When you refactor the function, the override is changed to an overload.

Note: If you remove a parameter, you need to manually remove any method code that uses the removed parameter.

Sync Edit Mode (Delphi, C#, C++)

Sync Edit mode allows you to change all occurrences of an identifier when you change one instance of that identifier. When you enter Sync Edit mode, you can tab to each highlighted identifier in your current **Code Editor** window. If you change an identifier that appears elsewhere in the file, all occurrences transform to whatever you type, character by character.

Undoing a Refactoring (Delphi, C#)

The refactoring engine takes advantage of a versioning mechanism, known as *local striping*, to allow you to undo renames in source code files. The IDE records the current timestamp of each file included in the current refactoring changeset. The timestamp corresponds to a specific local revision of the file. When you select the undo command, the IDE copies the local backup file that matches that timestamp back over the refactored file.

The important point to understand is that any changes that you make to the files after the refactoring will also be rolled back when you perform an Undo. Before the Undo is applied, you will get a warning message confirming that you want to apply the Undo. Applying the Undo reverts changes back to before the refactoring was originally applied in all modified files. You will lose any changes made in those files since the refactoring was originally applied.

Undo performs local striping only for Rename because Rename is the only refactoring operation that affects multiple files.

If you want to undo Extract Method, Declare Field, or Declare Variable refactorings, use Ctrl-z (regular Undo) in the **Code Editor**, or the **Undo** button in the **Refactoring** window, which accomplishes the same thing.

Testing Applications

Unit testing is an integral part of building reliable applications. The following topics discuss unit testing features included in Developer Studio 2006.

In This Section

Unit Testing Overview
Describes the integration of DUnit and NUnit in Developer Studio 2006.

Building Tests
Describes how to build tests with Unit Test Wizards.

Unit Testing Overview

Developer Studio 2006 integrates two open-source testing frameworks, DUnit and NUnit, that allow you to build and run automated test cases for your Delphi and C# applications. These frameworks simplify the process of building tests for classes and methods in your application. Using unit testing in combination with refactoring can improve your application stability. Testing a standard set of tests every time a small change is made throughout the code makes it more likely that you will catch any problems early in the development cycle.

The testing frameworks are both based on the JUnit test framework and share much of the same functionality.

This topic includes the following information:

- What Gets Installed.
- Test Projects.
- Test Cases.
- Test Fixtures.

What Gets Installed

Both products are installed during the complete Developer Studio 2006 installation. DUnit is installed by default, however, you can choose not to install NUnit or you can choose to install NUnit to a non-default location.

DUnit

DUnit gets installed automatically by the Developer Studio 2006 installer. You can find many DUnit resources in the \source\DUnit directory, under your primary installation directory. These resources include documentation and test examples.

When using DUnit, at a minimum you usually include at least one test case and one or more test fixtures. Test cases typically include one or more assertion statements to verify the functionality of the class being tested.

DUnit is licensed under the Mozilla Public License 1.0 (MPL).

NUnit

During the install process, you will be prompted to install NUnit. You can change the default location of the installation, or you can accept the default, which installs NUnit into C:\Program Files\NUnit V2.x, where *x* is a point release number.. The installation directory includes a number of resources including documentation and example tests.

NUnit is the name of the .NET testing framework and can be used with both Delphi for .NET and C# projects. There are some subtle but important differences between the way NUnit and DUnit work. For example, NUnit does not link in .dcu files, as DUnit does.

When using NUnit, at a minimum, you usually include at least one test case and one or more test fixtures. Test cases typically include one or more assertion statements to verify the functionality of the class being tested.

Test Projects

A test project encapsulates one or more test cases and is represented by a node in the IDE **Project Manager**. You can create a test project before creating test cases. Once you have a test project that is associated with a code project, you can add test cases to the test project. Developer Studio 2006 provides a **Test Project Wizard** to help you build a test project.

Test Cases

Every class that you want to test must have a corresponding test class. You define a test case as a class in order to instantiate test objects, which makes the tests easier to work with. You implement each test as a method that corresponds to one of the methods in your application. More than one test can be included in a test case. The ability to group and combine tests into test cases and test cases into test projects is what sets a test case apart from simple forms of testing, such as using print statements or evaluating debugger expressions. Each test case and test project is reusable and rerunnable, and can be automated through the use of shell scripts or console commands.

Generally, you should create your tests in a separate project from the source file project. That way, you do not have to go through the process of removing your tests from your production application. Developer Studio 2006 provides a **Test Case Wizard** to help you build test cases. You can add test cases directly into the same project as your source file, however, doing so increases the size of your project. You can also conditionally compile your test cases out of production code by using IFDEF statements around the test case code.

Test Fixtures

The term *test fixture* refers to the combination of multiple test cases, which test logically related functionality. You define test fixtures in your test case. Typically, you will instantiate your objects, initialize variables, set up database connection, and perform maintenance tasks in the SetUp and TearDown sections. As long as your tests all act upon the same objects, you can include a number of tests in any given test fixture.

DUnit Overview

DUnit is an open-source unit test framework based on the JUnit test framework. The DUnit framework allows you to build and execute tests against Delphi Win32 applications. The Developer Studio 2006 integration of DUnit allows you to test both Delphi Win32 and Delphi .NET applications.

Each testing framework provides its own set of methods for testing conditions. The methods represent common assertions. You can also create your own custom assertions. You will be able to use the provided methods to test a large number of conditions.

This topic includes information about:

- Building DUnit Tests.
- DUnit Functions.
- DUnit Test Runners.

Building DUnit Tests

Every DUnit test implements a class of type TTestCase. The following sample Delphi Win32 program defines two functions that perform simple addition and subtraction:

```
unit CalcUnit;

interface

type

{ TCalc }

  TCalc = class
  public
    function Add(x, y: Integer): Integer;
    function Sub(x, y: Integer): Integer;
  end;

implementation

{ TCalc }

function TCalc.Add(x, y: Integer): Integer;
begin
  Result := x + y;
end;

function TCalc.Sub(X, Y: Integer): Integer;
begin
  Result := x + y;
end;

end.
```

The following example shows the test case skeleton file that you need to modify to test the two functions, Add and Sub, in the preceding code.

```
unit TestCalcUnit;
```

```
interface

uses
  TestFramework, CalcUnit;
type
  // Test methods for class TCalc
  TestTCalc = class(TTestCase)
  strict private
    aTCalc: TCalc;
  public
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestAdd;
    procedure TestSub;
  end;

implementation

procedure TestTCalc.SetUp;
begin
  aTCalc := TCalc.Create;
end;

procedure TestTCalc.TearDown;
begin
  aTCalc := nil;
end;

procedure TestTCalc.TestAdd;
var
  _result: System.Integer;
  y: System.Integer;
  x: System.Integer;
begin
  _result := aTCalc.Add(x, y);
  // TODO: Add testcode here
end;

procedure TestTCalc.TestSub;
var
  _result: System.Integer;
  y: System.Integer;
  x: System.Integer;
begin
  _result := aTCalc.Sub(x, y);
  // TODO: Add testcode here
end;

initialization
  // Register any test cases with the test runner
  RegisterTest(TestTCalc.Suite);
end.
```

DUnit Functions

DUnit provides a number of functions that you can use in your tests.

Function	Description
Check	Checks to see if a condition was met.
CheckEquals	Checks to see that two items are equal.
CheckNotEquals	Checks to see if items are not equal.
CheckNotNull	Checks to see that an item is not null.
CheckNull	Checks to see that an item is null.
CheckSame	Checks to see that two items have the same value.
EqualsErrorMessage	Checks to see that an error message emitted by the application matches a specified error message.
Fail	Checks that a routine fails.
FailEquals	Checks to see that a failure equals a specified failure condition.
FailNotEquals	Checks to see that a failure condition does not equal a specified failure condition.
FailNotSame	Checks to see that two failure conditions are not the same.
NotEqualsErrorMessage	Checks to see that two error messages are not the same.
NotSameErrorMessage	Checks that one error message does not match a specified error message.

For more information on the syntax and usage of these and other DUnit functions, see the DUnit help files in \source \dunit\doc.

DUnit Test Runners

A test runner allows you to run your tests without impacting your application. The DUnit test project you create is your test runner. You can indicate the TextTestRunner to output test results to the console. The GUI test runner displays your results interactively in a GUI window right in the IDE. The results are color-coded to highlight which tests succeeded and which failed.

The GUI test runner is very useful when actively developing unit tests or the code you are testing. The GUI test runner displays a green bar over a test that completes successfully, a red bar over a test that fails, and a yellow bar over a test that is skipped.

The DUnit console/text test runner is useful when you need to run completed code and tests from automated build scripts.

NUnit Overview

NUnit is an open-source unit test framework based on the JUnit test framework. The NUnit framework allows you to build and execute tests against .NET Framework applications. The Developer Studio 2006 integration of NUnit allows you to test both Delphi for .NET and C# applications.

This topic includes information about:

- Building NUnit Tests.
- NUnit Asserts.
- NUnit Test Runners.

Building NUnit Tests

Each testing framework provides its own set of methods for testing conditions. The methods support common assertions. You can also create your own custom assertions. You will be able to use the provided methods to test a large number of conditions.

If you want to create tests for an application, you can first create a Test Project. The Test Project contains the Test Case files, which contain one or more tests. A test case is analogous to a class. Each test is analogous to a method. Typically, you might build one test for each method in your application. You can test each method in your application classes to make sure that the method performs the task you expect.

When you create a Test Project and add a Test Case to it, Developer Studio 2006 builds two template files: a test project template, which contains the attributes needed to compile the test project into an assembly, and a test case template, which contains the basic structure of the test case. The Test Case Wizard generates a skeleton test method for each method in the class being tested. This includes local variable declarations for each of the parameters to the method being called. You will need to write the code required to setup the parameters for the call (in SetUp) and the appropriate call to verify the return values or other state that is appropriate following the call (in TearDown).

The following example shows a small C# program that performs simple addition and subtraction:

```
[C#]
using System;

namespace CSharpCalcLib
{
    /// <summary>
    /// Simple Calculator Library
    /// </summary>
    public class Calc
    {
        public int Add(int x, int y)
        {
            return x + y;
        }

        public int Sub(int x, int y)
        {
            return x + y;
        }
    }
}
```

The following example shows the test case skeleton file that you need to modify to test the two methods, Add and Sub, in the preceding code.

```
[Delphi]
namespace TestCalc
{
    using System;
    using System.Collections;
    using System.ComponentModel;
    using System.Data;
    using NUnit.Framework;
    using CSharpCalcLib;


    // Test methods for class Calc
    [TestFixture]
    public class TestCalc
    {

        private Calc aCalc;

        [SetUp]
        public void SetUp()
        {
            aCalc = new Calc();
        }

        [TearDown]
        public void TearDown()
        {
            aCalc = null;
        }

        [Test]
        public void TestAdd()
        {
            int x;
            int y;
            int returnValue;
            // TODO: Setup call parameters
            returnValue = aCalc.Add(x, y);
            // TODO: Validate return value
        }

        [Test]
        public void TestSub()
        {
            int x;
            int y;
            int returnValue;
            // TODO: Setup call parameters
            returnValue = aCalc.Sub(x, y);
            // TODO: Validate return value
        }
    }
}
```

Note: Each test method is automatically decorated with the [Test] attribute in C# projects. In addition, in C# the test methods are defined as functions returning void.

The following example shows a small Delphi for .NET program that performs simple addition and subtraction:

```
unit CalcUnit;

// .Net Version

interface

type

{ TCalc }

  TCalc = class
  public
    function Add(x, y: Integer): Integer;
    function Sub(x, y: Integer): Integer;
  end;

implementation

{ TCalc }

function TCalc.Add(x, y: Integer): Integer;
begin
  Result := x + y;
end;

function TCalc.Sub(X, Y: Integer): Integer;
begin
  Result := x + y;
end;

end.
```

The following example shows the test case skeleton file that you need to modify to test the two functions, Add and Sub, in the preceding code.

```
unit TestCalcUnit;

interface

uses
  NUnit.Framework, CalcUnit;

type
  // Test methods for class TCalc
  [TestFixture]
  TestTCalc = class
  strict private
    FCalc: TCalc;
  public
    [SetUp]
    procedure SetUp;
    [TearDown]
    procedure TearDown;
  published
    [Test]
    procedure TestAdd;
    [Test]
    procedure TestSub;
  end;
```

```
implementation

procedure TestTCalc.SetUp;
begin
  FCalc := TCalc.Create;
end;

procedure TestTCalc.TearDown;
begin
  FCalc := nil;
end;

procedure TestTCalc.TestAdd;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // TODO: Setup call parameters
  ReturnValue := FCalc.Add(x, y);
  // TODO: Validate return value
end;

procedure TestTCalc.TestSub;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // TODO: Setup call parameters
  ReturnValue := FCalc.Sub(x, y);
  // TODO: Validate return value
end;

end.
```

Note: In Delphi for .NET the test methods are defined as procedures.

Each test method must:

- be public
- be a procedure for Delphi for .NET or a function with a void return type for C#
- take no arguments

Setup

Use the SetUp procedure to initialize variables or otherwise prepare your tests prior to running. For example, this is where you would set up a database connection, if needed by the test.

TearDown

The TearDown method can be used to clean up variable assignments, clear memory, or perform other maintenance tasks on your tests. For example, this is where you would close a database connection.

NUnit Asserts

NUnit provides a number of asserts that you can use in your tests.

Function	Description	Syntax
AreEqual	Checks to see that two items are equal.	`Assert.AreEqual(expected, actual [, string message])`
IsNull	Checks to see that an item is null.	`Assert.IsNull(object [, string message])`
IsNotNull	Checks to see that an item is not null.	`Assert.IsNotNull(object [, string message])`
AreSame	Checks to see that two items are the same.	`Assert.AreSame(expected, actual [, string message])`
IsTrue	Checks to see that an item is **True**.	`Assert.IsTrue(bool condition [, string message])`
IsFalse	Checks to see that an item is **False**.	`Assert.IsFalse(bool condition [, string message])`
Fail	Fails the test.	`Assert.Fail([string message])`

You can use multiple asserts in any test method. This collection of asserts should test the common functionality of a given method. If an assert fails, the entire test method fails and any other assertions in the method are ignored. Once you fix the failing test and rerun your tests, the other assertions will be executed, unless one of them fails.

NUnit Test Runners

A test runner allows you to run your tests without impacting your application. If you use the console test runner, it directs the output to the console. If you use the GUI test runner, you can see the results interactively in a GUI non-modal window right in the IDE. The results are color-coded to highlight which tests succeeded and which failed.

NUnit includes two test runners:

- NUnitConsole.exe
- NUnitGUI.exe

The GUI test runner is very useful when actively developing unit tests or the code you are testing. The GUI test runner displays a green bar over a test that completes successfully, a red bar over a test that fails, and a yellow bar over a test that is skipped.

The NUnit console test runner is useful when you need to run completed code and tests from automated build scripts. If you want to redirect the output to a file, use the redirection command parameter. The following example shows how to redirect test results to a TestResult.txt text file:

```
nunit-console nunit.tests.dll /out:TestResult.txt
```

Note: You may need to set the path to your host application in the **Project Options** dialog. Set the Host Application property to the location of the test runner you want to use.

Localizing Applications

Developer Studio 2006 includes a suite of Translation Tools to facilitate localization and development of .NET and Win32 applications for different locales. The Translation Tools include the following:

- Satellite Assembly Wizard (for .NET)
- Resource DLL Wizard (for Win32)
- Translation Manager
- Translation Repository

The Translation Tools are available for Delphi VCL Forms applications (both .NET and Win32), and Win32 console applications, packages, and DLLs. You can access the Translation Tools configuration options by choosing **Tools** ▸ **Options** ▸ **Translation Tools Options**.

The Wizards

Before you can use the Translation Manager or Translation Repository, you must add languages to your project by running either the Satellite Assembly Wizard for .NET projects or the Resource DLL Wizard for Win32 projects. The Satellite Assembly Wizard creates a .NET satellite assembly for each language you add. The Resource DLL Wizard creates a Win32 resource DLL for each language. For simplicity, this documentation uses the term *resource module* to refer to either a satellite assembly or a resource DLL.

While running either wizard, you can include extra files, such as .resx or .rc files, that are not normally part of a project. You can add new resource modules to a project at any time. If you have multiple projects open in the IDE, you can process several at once.

You can also use the wizards to remove languages from a project and restoring languages to a project.

Translation Manager

After resource modules have been added to your project, you can use the Translation Manager to view and edit VCL forms and resource strings. After modifying your translations, you can update all of your application's resource modules.

The External Translation Manager (ETM) is a version of the Translation Manager that you can set up and use without the IDE. ETM has the same functionality as the Translation Manager, with some additional menus and toolbars.

Translation Repository

The Translation Repository provides a database for translations that can be shared across projects, by different developers. While working in the Translation Manager, you can store translated strings in the Repository and retrieve translated strings from the Repository.

By default, each time your assemblies are updated, they will be populated with translations for any matching strings that exist in the Repository. You can also access the Repository directly, through its own interface, to find, edit, or delete strings.

The Translation Repository stores data in XML format. By default, the file is named default.tmx and is located in the Developer Studio 2006\bin directory.

Files Generated by the Translation Tools

The files generated by the Translation Tools include the following:

File extension	Description
.nfn (.NET) .dfn (Win32)	The Translation Tools maintain a separate file for each form in your application and each target language. These files contain the data (including translated strings) that you see in the Translation Manager.
.resx (.NET)	The Satellite Assembly Wizard uses the compiler-generated .drcil file to create an .resx file for each target language. These .resx files contain special comments that are used by the Translation Tools.
.rc (Win32)	The Resource DLL Wizard uses the compiler-generated .drc file to create an .resx file for each target language. These .resx files contain special comments that are used by the Translation Tools.
.tmx	The Translation Repository stores data in an .tmx file. You can maintain more than one repository by saving multiple .tmx files.
.bdsproj	The External Translation Manager lists the assemblies (languages) and resources to be translated into a .bdsproj project file. When third-party translators add and remove languages from a project, they can save these changes in an .bdsproj file, which they return to the developer.

Note: You should not edit any of these files manually.

Debugging Applications

Many of the same techniques are used for debugging applications in different environments. Developer Studio 2006 provides an integrated debugging environment that enables you to debug Win32 application and .NET applications. In addition, you can use the debugger to debug an application running on a remote machine that does not have Developer Studio 2006 installed.

In This Section

Overview of Debugging

Provides general debugging information and describes the debugging tools available in Developer Studio 2006.

Overview of Remote Debugging

Provides an overview of debugging an application on a remote machine that does not have the IDE installed.

Overview of Debugging

Developer Studio 2006 includes both the Borland .NET Debugger and Borland Win32 Debugger. The IDE automatically uses the appropriate debugger based on the active project type. Cross-platform debugging within a project group is supported and, where possible, the debuggers share a common user interface.

The integrated debuggers let you find and fix both runtime errors and logic errors in your Developer Studio 2006 application. Using the debuggers, you can step through code, set breakpoints and watches, and inspect and modify program values. As you debug your application, the debug windows are available to help you manage the debug session and provide information about the state of your application.

Stepping Through Code

Stepping through code lets you run your program one line of code at a time. After each step, you can examine the state of the program, view the program output, modify program data values, and continue executing the next line of code. The next line of code does not execute until you tell the debugger to continue.

The **Run** menu provides the **Trace Into** and **Step Over** commands. Both commands tell the debugger to execute the next line of code. However, if the line contains a function call, **Trace Into** executes the function and stops at the first line of code *inside* the function. **Step Over** executes the function, then stops at the first line *after* the function.

Evaluate/Modify

The Evaluate/Modify functionality allows you to evaluate an expression. You can also modify a value for a variable and insert that value into the variable. The Evaluate/Modify functionality is customized for the language you are using. For a C++ project, the Evaluate/Modify dialog accepts only C++ expressions. For a C# project, the Evaluate/Modify dialog accepts only C# expressions. For a Delphi project, the Evaluate/Modify dialog accepts only Delphi expressions.

Breakpoints

Breakpoints pause program execution at a certain point in the program or when a particular condition occurs. You can then use the debugger to view the state of your program, or step over or trace into your code one line or machine instruction at a time. The debugger supports three types of breakpoints. Source breakpoints pause execution at a specified location in your source code. Address breakpoints pause execution at a specified memory address. Data breakpoints allow you to pause execution when memory at a particular address changes.

Note: Data breakpoints are available only for the Win32 debugger.

Watches

Watches lets you track the values of program variables or expressions as you step over or trace into your code. As you step through your program, the value of the watch expression changes if your program updates any of the variables contained in the watch expression.

Debug Windows

The following debug windows are available to help you debug your program. By default, most of the windows are displayed automatically when you start a debugging session. You can also view the windows individually by using the **View** ▸ **Debug Windows** sub-menu.

Each window provides one or more right-click context menus. The F1 Help for each window provides detailed information about the window and the context menus.

Debug Window	Description
Breakpoint List	Displays all of the breakpoints currently set in the **Code Editor** or **CPU** window.
Call Stack	Displays the current sequence of function calls.
Watch List	Displays the current value of watch expressions based on the scope of the execution point.
Local Variables	Displays the current function's local variables, enabling you to monitor how your program updates the values of variables as the program runs.
Modules	Displays processes under control of the debugger and the modules currently loaded by each process. It also provides a hierarchical view of the namespaces, classes, and methods used in the application.
Threads Status	Displays the status of all processes and threads of execution that are executing in each application being debugged. This is helpful when debugging multi-threaded applications.
Event Log	Displays messages that pertain to process control, breakpoints, output, threads, and module.
CPU	Displays the low-level state of your program, including the assembly instructions for each line of source code and the contents of certain registers.
FPU	Displays the contents of the Floating-point Unit and SSE registers in the CPU.

Remote Debugging

Remote debugging lets you debug an application running on a remote computer. Your computer must be connected to the remote computer through TCP/IP and the remote debugger must be installed on the remote machine. After you create and copy the required application files to the remote computer, you can connect to that computer and begin debugging.

Overview of Remote Debugging

Remote debugging enables you to debug one or more applications on a remote machine when the IDE is running only on your local machine. This allows debugging on a machine where it is impractical to install the entire IDE and rebuild a project. Remote debugging is useful for applications that run differently on your local machine than on an end user's machine.

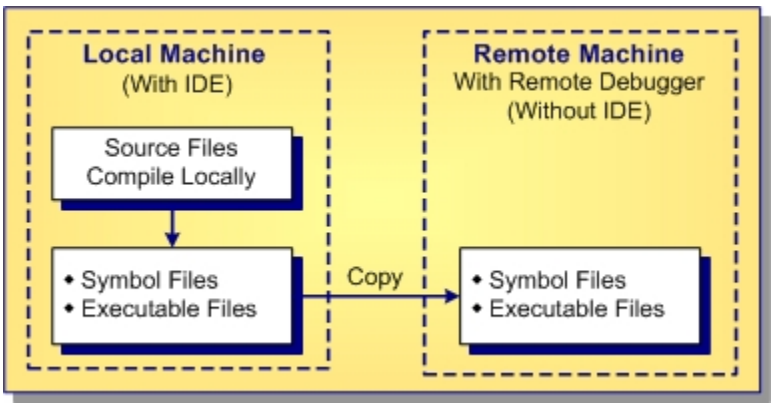
The Remote Debugger Executable

The remote debugger executable is named rmtdbg100.exe. The executable and its supporting files must be present on the remote machine. The easiest way to install the executable is directly from the Developer Studio 2006 installation disk. However, if the installation disk is not available, you can copy the required files from a machine that has the full Developer Studio 2006 IDE installed.

Local and Remote Files

Three types of files are involved in remote debugging:

- Source files
- Executable files
- Symbol files



Source files are compiled using the IDE on the local machine. The executable files and symbol files produced after compilation must be copied to the remote machine.

Source Files

When you debug a project on a remote machine, the source files for the project must be open on the local machine. The source files display in the editor window to show a program's current execution point. You do not use source files on the remote machine.

Executable Files

Executable files are the .dll files and .exe files that are mapped into the application's address space. You generate these files on the local machine, then copy them to the remote machine.

Symbol Files

Symbol files are generated on the local machine at compile time. These are used by the debugger to get information such as the mapping of machine instructions to source line numbers or the names and types of variables declared in the source files. The extension for the symbol files depends on the language, as shown in the following table:

Language	Debug symbol file extension
Delphi for Win32	.rsm
Delphi for .NET	.rsm and .pdb
C++	.tds
C#	.pdb

You must set up specific options to generate symbol files on the local machine, then copy the files to the remote machine.

Local and Remote Machines

To use remote debugging, you must be able to log on to the remote machine and you must have write access to at least one directory.

Note: The remote debugger does not provide a mechanism for interacting with an application on the remote machine. If you need to interact with the application, you must establish a remote desktop connection.

Deploying Applications

After you have written, tested, and debugged your application, you can make it available to others by deploying it. Depending on the size and complexity of the application, you can package it as one or more assemblies, as compressed cabinet (.cab) files, or in an installer program format (such as .msi). After the application is packaged, you can distribute it by using XCOPY, FTP, as a download, or with an installer program.

This sections includes the following general topics:

- Deploying .NET Applications
- Deploying Win32 Applications
- Using Installation Programs
- Redistributing Developer Studio 2006 Files
- Redistributing Third Party Software

For additional information about deploying specific types of applications, refer to the list of links at the end of this topic.

Deploying .NET Applications

Assuming that the target computer already has the .NET Framework installed on it, deploying a simple application that consists of a single executable is as easy as copying the .exe file to the target computer. You don't need to register the application and deleting the application files effectively uninstalls it.

Applications That Include Shared Assemblies

If your application includes an assembly that will be shared by other applications, you will need to uniquely identify the assembly with a strong name and then install it in the Global Assembly Cache (GAC). The strong name consists of the assembly's text name, version number, optional culture information, and the public key and digital signature to ensure uniqueness. The .NET Framework SDK provides command line utilities for creating a public/private key (sn.exe), assigning a strong name (al.exe), and installing an assembly in the GAC (gacutil.exe). For more information about these utilities, see the Framework SDK online Help.

Deploying VCL.NET Applications

When building applications that use the VCL .NET framework, the way you build the application determines what files you need to distribute with it. If you build the application by compiling VCL for .NET units directly into the program executable file, the application will have external dependencies only on the .NET Framework.

However, if you build the application by compiling the application to have external references to VCL for .NET assemblies, the application will have external dependencies on the .NET Framework, the Borland.Delphi.dll, and whatever Developer Studio 2006 packages you have added to the project references, for example, Borland.VclRtl.dll or Borland.Vcl.dll.

Deploying ASP.NET Applications

Developer Studio 2006 includes the ASP.NET Deployment Manager to assist you in deploying ASP.NET applications. You can use it to deploy to a remote computer by using a share or an FTP connection, or to your local computer. When you add a Deployment Manager to your project, an XML file (.bdsdeploy) is added to the project directory and a **Deploy** tab is added to the IDE. You provide destination and connection information on the **Deploy** tab and optionally modify the suggested list of files to copy, then the Deployment Manager copies the files to the deployment destination.

Redistributing the .NET Framework

If you plan to deploy your application to a computer that does not have the .NET Framework installed on it, you will need to redistribute and install the .NET Framework with your application. Microsoft provides a redistributable

installer called dotnetfx.exe, which contains the common language runtime and .NET Framework components required to run .NET applications. For more information about dotnetfx.exe, see the .NET Framework SDK online Help.

Before Deploying a C# Application

Typically, while developing a C# application, you compile it with debugging information to facilitate testing. When you create a new project, it uses the default **Debug** option set, which creates the executable files and the program database file (.pdb) for debugging in the *project*\bin\Debug directory.

When you are ready to deploy the C# application, you can compile it using the default or a user-defined **Release** option set to create an optimized version of the application in the *project*\bin\Release directory. The optimized application is smaller, faster, and more efficient. To change the **Debug/Release** option sets, choose **Project ▸ Options**.

Deploying Win32 Applications

For information on deploying Win32 applications, refer to the **Deploying Win32 Applications** link at the end of this topic.

Using Installation Programs

For complex applications that consist of multiple files, you can use an installation program. Installation programs perform various tasks, such as copying executable and supporting files to the target computer and making Windows registry entries.

Setup toolkits, such as InstallShield Express, automate the process of creating installation programs, often without the need to write any code. InstallShield Express is based on Windows Installer (MSI) technology and can be installed from the Developer Studio 2006 installation CD. After installing it, refer to the online InstallShield online Help for information about using the product.

Redistributing Developer Studio 2006 Files

Many of the files associated with Developer Studio 2006 applications are subject to redistribution limitations or cannot be redistributed at all. Refer to the following documents for the legal stipulations regarding the redistribution of these files.

File	Description
deploy.htm	Contains deployment considerations for each edition of Developer Studio 2006.
license.txt	Addresses legal rights and obligations concerning Developer Studio 2006.
readme.htm	Contains last minute information about Developer Studio 2006, possibly including information that could affect the redistribution rights for Developer Studio 2006 files.

These files are located, by default, at C:\Program Files\Borland\BDS\4.0.

Redistributing Third Party Software

The redistribution rights for third party software, such as components, utilities, and helper applications, are governed by the vendor that supplies the software. Before you redistribute any third party software with your Developer Studio 2006 application, consult the third party vendor or software documentation for information regarding redistribution.

Procedures

Getting Started Procedures

Adding and Removing Files

You can add and remove a variety of file types to your projects.

To add a file to a project

1. Choose **Project** ▸ **Add to Project**.
   The **Add to Project** dialog box appears.
2. Select a file to add and click **Open**.
   The file appears below the Project.exe node of the **Project Manager**.

To remove a file from a project

1. Choose **Project** ▸ **Remove From Project**.
   A **Remove From Project** dialog box appears.
2. Select the file or files you want to remove and click **OK**.

Adding Components to a Form

To add components to a form

1. On the **Tool Palette**, select a visual or nonvisual component.
2. Double-click the component to place it on the form or drag the component onto the form.
   If you add a nonvisual component to the form, the component tray appears at the bottom of the Designer surface.
3. Repeat steps 1 and 2 to add additional components.
4. Use the dotted grid on the form to align your components.

Adding References

You can integrate your legacy COM servers and ActiveX controls into managed applications by adding references to unmanaged DLLs to your project, and then browse the types just as you would with managed assemblies.

To add references

1. From the main menu, choose **Project** ▸ **Add Reference**.

   The **Add Reference** dialog box appears.

2. Select either a legacy COM type library or ActiveX control to integrate into your managed application.
3. Click **Add Reference**.

   The reference is added to the text box.

4. Click **OK**.

> **Tip:** You can also right-click the **References** folder in the **Project Manager**, and choose **Add Reference**.

Adding Templates to the Object Repository

You can add your own objects to the **Object Repository** as templates to reuse or share with other developers. Reusing objects lets you build families of applications with common user interfaces and functionality to reduce development time and improve quality.

To add a template to the Object Repository

1. Save your project.
2. Choose **Project** ▸ **Add to Repository**.
3. Enter the project name, description, and author information in the dialog box.
4. Click **Browse** to select an icon to represent the project you saved.
5. Click **OK**.

Configuring Together

Together is flexibly configurable. Use the **Options** dialog window to tune modeling features to best fit your requirements.

The **Options** dialog window provides a number of diagram customization settings. You can configure the appearance and layout of the diagrams, specify font properties, member format, and level of detail.

To configure Together settings:

1. On the main menu, choose **Tools** ▸ **Options**.
2. In the **Options** dialog window, expand the Together category.
3. Select the desired option level.
4. For the Project and Diagram option levels, choose the project or diagram where the configuration changes should apply. To do that, click the chooser buttons in the corresponding fields and select the desired project or diagram from the model.
5. Click the desired subcategory.
6. Edit configuration options as required.
7. Click **OK** to apply changes and close the dialog window.

You can make configuration options final at a certain parent level and disable any changes on the lower levels:

To disable configuration changes:

1. On the main menu, choose **Tools** ▸ **Options**.
2. Click the Together category to expand it.
3. Select the required sub-category (default, project group or project).
4. Check the **Disable sublevels** option.

Copying References to a Local Path

During runtime, assemblies must be in the output path of the project or in the GAC for deployment. If your project contains a reference to an object that is not in one of the two locations, the reference must be copied to the appropriate output path.

To a copy reference to a local path

1. In the **Project Manager**, right-click an assembly DLL in the **References** folder.
2. Set the **Copy Local** option to copy the file to the output directory.

Note: The IDE maintains the **Copy Local** setting until you change it.

Creating a Component Template

You can save selected, preconfigured components on the current form as a reusable component template accessible from the **Tool Palette**.

To create a component template

1. Place and arrange components on a form.
2. In the **Object Inspector**, set the component properties and events as desired.
3. Select the components that you want to save as a component template. To select several components, drag the mouse over them.

   Tip: To select all of the components on the form, choose **Edit** ▸ **Select All**.

   Gray handles appear at the corners of each selected component.
4. Choose **Component** ▸ **Create Component Template**.

   The **Create Component Template** dialog box appears.
5. Specify a name, a **Tool Palette** category, and an icon for the template.
6. Click **OK**.

Your new template appears immediately on the **Tool Palette**, in the category that you specified.

To use a component template

1. Display the form to which you want to add the components from the component template.
2. On the **Tool Palette**, double-click the component template icon.

   The components in the component template are added to the form, along with their preconfigured properties and events. You can reposition the components independently, reset their properties, and create or modify event handlers for them, just as if you had placed each component in a separate operation.

To delete a component template

1. On the **Tool Palette**, right-click the component template to display a context menu.
2. Choose the **Delete [template name] Button** command.

   The component template is deleted immediately from the **Tool Palette**.

Creating a Project

To add a new project

1. Choose **Project** ▸ **Add New Project**.
   The **New Items** dialog box appears.
2. Select a project and click **OK**.
   The project is added to the **Project Manager**.

To add an existing project

1. Choose **Project** ▸ **Add Existing Project**.
   The **Open Project** dialog box appears.
2. Select an existing project to add and click **Open**.

Customizing the Form

To customize the form

1. Choose **Tools** ▸ **Options**.
2. From the **Options** dialog box, click **Windows Forms Designer**.
3. Enable or disable the snap to grid and show grid features by selecting and deselecting the check boxes.
4. Choose one of the bracing styles.
5. Click **OK**.

Tip: The changes will affect only forms created after these options are changed. To change the settings for existing forms, set the GridSize, DrawGrid, and SnapToGrid properties of the form.

Customizing the Tool Palette

To arrange individual components

1. Click the component.
2. Drag the component anywhere within the **Tool Palette**.

To arrange an entire category of components

1. Click a category name .
2. Drag the category anywhere within the **Tool Palette**.
3. Release your mouse button to place the category in the desired location.

To add additional categories

1. Right-click the **Tool Palette**.
2. Choose the **Add New Category** command.
   The **Create a new Category** dialog box appears.
3. Enter a name for the category in the **New Category Name** text box.
4. Click **OK**.
   The new category appears at the bottom of the **Tool Palette**.

Customizing Toolbars

To arrange your toolbars

1. Click the grab bar on the left side of any toolbar.
2. Drag the toolbar to another location or onto your desktop.

To delete icons from the toolbar

1. Choose **View** ▸ **Toolbars** ▸ **Customize**.
2. From the toolbar, not the **Customize** dialog box, drag the tool from the toolbar until its icon displays an X and then release the mouse button.
3. When completed, click **Close**.

To add icons to the toolbar

1. Choose **View** ▸ **Toolbars** ▸ **Customize**.
2. Click the **Commands** tab.
3. In the **Categories** list, select a category to view its tool icons.
4. From the **Commands** list, drag the selected icon onto the toolbar of your choice.
5. When completed, click **Close**.

Docking Tool Windows

The Auto-Hide feature lets you undock and hide tool windows, such as the **Object Inspector**, **Tool Palette**, and **Project Manager**, but still have access to them.

To use Auto-Hide to hide your tools

1. Click the push pin in the upper right corner of a tool window.
   The tool window is replaced by one or more tabs at the outer edge of the IDE window.
2. To display the tool window, position the cursor over the tab.
   The tool window slides into view.
3. To slide the tool window out of view, move the cursor away from the tool window.
4. To redock the tool window, click the push pin until it points down.

To dock the tools with one another

1. Click the tool window title bar and drag the window into another tool window.
2. Select a location to drop the tool window and release the mouse button.

To undock the tools from one another

1. Click the tool window title bar and drag the window away from the other tool window.
2. Select a location to drop the tool window and release the mouse button.

Exploring .NET Assembly Metadata

You can open and explore the namespaces and types contained with a .NET assembly. The assembly metadata is displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the namespaces and types within the assembly. The right pane displays specific information on the selected item in the tree. The **Call Graph** tab shows you a list of the methods called by the selected method, as well as a list of the methods that call the selected method.

To inspect a .NET assembly

1. Choose **File** ▸ **Open**.
2. In the **Open** dialog box, from the **Files of type** drop-down list, select **Assembly Metadata**.
3. Navigate to the folder where the .NET assembly is located. Select the assembly and click **Open**.

You can open multiple .NET assemblies in the metadata explorer. Each open assembly is displayed in the tree in the left-pane; the top-level node for a .NET assembly is denoted by the clr icon.

To close a particular .NET assembly, right-click on the top-level clr icon and select **Close**.

Using the Call Graph tab

1. Select a method node in the left pane.
2. Select the **Call Graph** tab.

   The top half of the **Call Graph** tab shows you a list of methods that call the method you selected in the left pane.

   The bottom half of the **Call Graph** tab shows you the methods called by the method you selected in the left pane.

   Methods that exist in the same assembly as the currently selected method will appear as clickable links, and are displayed in blue underlined text. Clicking on a link will cause that method to become selected in the tree in the left-hand pane.

   Tip: You can use the Browser buttons on the toolbar to navigate backwards and forwards to previously selected items in the left pane.

Exploring Windows Type Libraries

You can open and inspect the interfaces and other types contained within a Windows type library. The type library contents are displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the interface and type definitions within the type library. The right pane displays specific information on the selected item in the tree. The **Type Library Explorer** can open a .TLB file, as well as OCX controls, and .DLL and .EXE files that have type libraries as embedded resources.

To Inspect a Windows Type Library

1. Choose **File** ▸ **Open**.
2. In the **Open** dialog box, from the **Files of type** drop-down list, select **Type Library**.
   This sets the file filter to display files with extensions of .TLB, .OLB, .OCX, .DLL, and .EXE.
3. Navigate to the folder where the type library is located.
4. Select the file and click **Open**.

You can open multiple type libraries in the explorer. Each open type library is displayed in the tree in the left pane; the top-level node for a type library is denoted by the icon.

To close a particular type library, right-click on the top-level icon and select **Close**.

Finding Items on the Tool Palette

To find items on the Tool Palette

1. Click anywhere on the **Tool Palette** and start typing the name of the item that you want to find.

   The **Tool Palette** is filtered to display only those item names that match what you are typing. The characters that you have typed appear in **bold** in the item names.

2. Double-click an item to perform the default action for that item. For example, double-clicking a component adds it to your form, whereas double-clicking a code snippet adds it to your code.

3. To remove the search filter from the **Tool Palette**, click the filter icon .

Installing Custom Components

To install custom components

1. Choose **Component** ▸ **Installed .NET Components**.
2. Click **Select an Assembly**.
3. Navigate to the folder containing the component assembly.

   Alternatively, you can enter the name of the full path to the assembly in the **File Name** field.
4. Select the assembly.
5. Click **Open**.

   The **Installed .NET Components** dialog box displays the components from the assembly.
6. Verify that the components you want to install on the **Tool Palette** are checked.
7. Click **OK**.

Installing More Computer Languages

If you have installed Developer Studio 2006 with only one or two computer languages (Delphi, C#, C++), and you later decide to add a language that was not originally installed, follow the steps below.

To add more computer languages to your IDE:

1. Choose **Start** ▸ **Settings** ▸ **Control Panel** ▸ **Add or Remove Programs**.
2. Select Developer Studio 2006
3. Click the **Change** button.
4. When the Installation Wizard comes up, it will ask you if you want to Modify, Repair, or Remove the program. Select the **Modify** radio button.
5. Follow the rest of the steps in the Installation Wizard to choose the languages that you want to add.
6. Click the **Finish** button.

Renaming Files Using the Project Manager

Renaming a file changes the name of the file in both the **Project Manager** and on disk.

To rename a file

1. In the **Project Manager**, right-click the file that you want to rename.

   The context menu is displayed.

2. Choose **Rename**.

3. Enter the new name for the file.

   If the file has associated files that appear as child nodes in the **Project Manager** tree, those files are automatically renamed.

Saving Desktop Layouts

You can switch between multiple desktop layouts. Choose a layout from the drop-down list box located on the **Desktop** toolbar. Additionally, you can save your desktop or debug desktop layouts as default.

To save a desktop layout

1. Choose **View** ▸ **Desktops** ▸ **Save Desktop**.
2. Enter the name of the desktop in the **Save Desktop** dialog box.
3. Click **OK**.

To set a debug desktop layout

1. Choose **View** ▸ **Desktops** ▸ **Set Debug Desktop**.
2. Select a debug desktop layout.
3. Click **OK**.

Setting Component Properties

After you place your components on your Designer, set their properties using the **Object Inspector**. By setting a component's properties, you can change the way a component appears and behaves in your application. Because properties appear during designtime, you have more control over a component's properties and can easily modify them without having to write additional code.

To set component properties

1. On the **Object Inspector**, click the **Properties** tab.
2. Set the component properties by entering values in the text box or through an editor.

   Boolean properties like **True** and **False** can be toggled.

Setting Dynamic Properties

Many of the .NET Framework objects support dynamic properties. Dynamic properties provide a way to change property values without recompiling an application. The dynamic properties and their values are stored in a configuration file, along with the application's executable file. Changing a property value in the configuration file causes the change to take effect the next time the applications runs. Dynamic properties are useful for changing an application after it has been deployed.

To set a dynamic property in the Object Inspector

1. In a form on the **Design** tab, click the object for which you want to set dynamic properties.
2. In the **Object Inspector**, expand **(DynamicProperties)** and click **(Advanced)**. If the object does not support dynamic properties, **(DynamicProperties)** is not displayed.

   Tip: If the **Object Inspector** is arranged by category, **(DynamicProperties)** is displayed under **Configurations**.

3. Click the **ellipsis (...)** button next to **(Advanced)** to display the **Dynamic Properties** dialog box.

   This dialog lists all of the properties that can be stored in the configuration file.
4. Select the properties you want to store in the configuration file.
5. Optionally, you can override the default key name listed in the **Key mapping** field.
6. Click **OK**.

   The dynamic properties are marked with an icon in the **Object Inspector**.

   Developer Studio 2006 creates an XML file named app.config (for a Windows application) or Web.config (for a Web application) in the project directory. This file lists the dynamic properties and their current values.
7. Compile the application.

   Developer Studio 2006 creates a file named <projectname>.exe.config (for a Windows application) or <projectname>.dll.config (for a Web application) in the same directory as the application's executable or DLL file.

To change a dynamic property value in the configuration file

1. In the directory that contains the application's executable or DLL file, locate the configuration file.
2. Open the file in a text editor.
3. Locate the add key= statement for the property to be changed and edit the value.
4. Save your changes and close the file.

The next time the application runs, the changed property value will be in effect.

Setting Project Options

You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.

To change compiler options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Select **Compiler** and set your options to modify how you want your program to compile.
3. Click **OK**.

To change application options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Select **Application** and specify a title and extension for your application.
3. Click **OK**.

To change debugger options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Use the **Debugger** page to pass command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger.
3. Use the **Environment Block** page to indicate which environment variables are passed to your application while you are debugging it.
4. Click **OK**.

Setting Properties and Events

Properties, methods, and events are attributes of a component.

To set object properties

1. On your form, click once on the object to select it.
2. In the **Object Inspector**, click the **Properties** tab.
3. Select the property that you want to change and either enter a value in the text box, select a value from the drop-down list, or click the ellipsis (...) next to the text box to use the associated property editor, depending on which update technique is available for the property.

To set an event handler

1. On your form, click once on the object to select it.
2. On the **Object Inspector**, click the **Events** tab.
3. If an event handler already exists, select it from the drop-down box. Otherwise, double-click the event to switch to **Code** view.
4. Type the code you want to execute when the event occurs.

Setting The IDE To Mimic Delphi 7

Use this procedure to set the IDE to mimic Delphi 7 or C++Builder, where each pane is its own window.

To turn off the Embedded Designer layout

1. Choose **Tools** ▸ **Options** ▸ **Environment Options** ▸ **VCL Designer**.
2. Uncheck **Embedded Designer**.
3. Click **OK**.
4. Restart Developer Studio 2006 for the change to take effect.

Setting Tool Preferences

You can customize the appearance and behavior of many tools and features, such as the **Object Inspector**, **Code Editor**, and integrated debugger.

To set tool preferences

1. Choose **Tools** ▸ **Options**.
2. Review the options in each tool category and customize the settings to suit your needs.
3. Click **OK**.

Using Design Guidelines with VCL Components

You can use VCL or VCL.NET (with Delphi or C++) to setup components that are "aware" of their relation to other components on a form. You can set properties to specify the distance between controls, shortcuts, focus labels, tab order, and maximum number of items (listboxes, menus).

To see and use the design guidelines:

1. Register an object type.
2. Indicate various points on or near a component's bounds that are "alignment" points. These "alignment" points are vertical or horizontal lines that cut across a visual control's bounds.
3. Supply UI guideline information so that each component will adhere to rules such as distance between controls, shortcuts, focus labels, tab order, maximum number of items (listboxes, menus),

Your new Error Reconcile Form will display four columns in the upper portion of the window, and six radio buttons in the bottom portion of the window. The following table describes each of the columns.

Component	Default Value when 'Use Design Guidelines' is Set
Alignment	The names of the columns of the table in which an error has occurred.
Margins	Bottom = 3, Left = 3, Right = 3, Right = 3, Top = 3
Padding	The last update that was saved to the Server. (This represents what the row contains on the server.)

Using Online Help

To get assistance while you work, do one of the following:

1. To see a description of what any screen element does in any opened dialog box, press F1 or click **Help**.
2. To see a relevant help topic for a pane, view, **Tool Palette** icon or another element, press F1.
3. To open the Table of Contents for online help, choose **Help** ▸ **Borland Help** on the main menu to see the **Contents** tab.
4. To search for specific topics and terms, use the **Index** tab.
5. If you have questions about Developer Studio 2006, visit Borland Technical Support at http://support.borland.com.

To filter help information, do the following:

1. To filter out the unnecessary books and topics from the Table of Contents and Index, choose one of the following filters in the Filtered by list box:
   - Developer Studio 2006 for .NET
   - Developer Studio 2006 for Win32
   - and so on
2. If a topic provides information that can be relevant to one or another Developer Studio 2006 feature set, you can show or hide the desired contents within a topic using the filter button.

Using To-Do Lists

A to-do list records and displays tasks that need to be completed for a project.

To create a to-do list and add an item to it

1. Choose **View** ▸ **To-Do List**.
2. In the **To-Do List** dialog box, right-click and choose **Add**.
3. In the **Add To-Do Item** dialog box, enter a description of the task and adjust the other fields as necessary.
4. Click **OK**.

To add a to-do list item as a comment in code

1. In the **Code Editor**, position your cursor where you want to add the comment.
2. Right-click and choose **Add To-Do List Item**.
3. In the **Add To-Do Item** dialog box, select the item that you want to add.
4. Click **OK**.

The item is added as a comment to your code, beginning with the word TODO.

To mark a to-do list item as completed

1. Choose **View** ▸ **To-Do List**.
2. In the **To-Do List** dialog box, check the check box next to the item to indicate completion.

   The item remains in the list, but the text is crossed out. If the item was added as a comment to code, the comment is updated to indicate DONE instead of TODO.

To filter the items in a to-do list

1. Choose **View** ▸ **To-Do List**.
2. Right-click the **To-Do List** dialog box and choose **Filter**.
3. Choose either **Categories**, **Owner**, or **Item types**, depending on which you want to filter.
4. In the **Filter To-Do List** dialog box, uncheck the items that you want to hide in the to-do list.
5. Click **OK**.

   The to-do list is redisplayed, with the filtered items hidden. The status bar at the bottom of the **To-Do List** dialog box indicates how many items are hidden due to filtering.

To delete an item from a to-do list

1. Choose **View** ▸ **To-Do List**.
2. In the **To-Do List** dialog box, select the item to delete.
3. Right-click and choose **Delete**.

   The item is removed from the **to-do list**. If the item was added as a comment to code, the comment is also removed.

Writing Event Handlers

Your source code usually responds to events that might occur to a component at runtime, such as a user clicking a button or choosing a menu command. The code that responds to an occurrence is called an event handler. The event handler code can modify property values and call methods.

To write an event handler

1. On your form, click the component for which you want to write an event handler.
2. To create the default event for the component, double-click the component on the form.

   To choose another event for the component, click the **Events** tab in the **Object Inspector**, locate the event, and double-click its text box.

   The **Code Editor** appears.
3. Type the code that will execute when the event occurs at runtime.

CaliberRM Procedures

Adding a Document Reference

Document references provide additional information for a requirement. You can add additional information as a text reference for a requirement, or add a web reference for the requirement.

To add a document reference for a requirement

1. Select the requirement to add a reference to.
2. Click the **References** tab. The tab is displayed.
3. Click **New Text** or **New Web**, depending on what to create.
4. In the pane at the bottom of the tab, enter the information for the reference. To view a web reference, double-click the web reference.
5. Click **Save** to save the changes or **Cancel Changes** to cancel the changes.
6. To remove a reference, select the reference and click **Remove**.

Adding a Table into a Requirement Description

To add a table to a requirement description

1. Select the requirement whose description in which to add a table.
2. Place your cursor where to insert the table and click **Insert Image**. The **Open** dialog box displays.
3. Indicate how many rows and columns to add.
4. Indicate the width and height, in pixels or as a percentage of the window.
5. Indicate how to align the table, the border width, the size of the cell padding (the amount of space between the contents of a table cell and the inside edges of a table cell; adding extra padding can prevent tables, especially large ones, from looking dense and crowded), and the size of the space between cells.
6. Click **OK** to add the table, or **Cancel** to cancel the addition.

   To delete the table, place your cursor at a corner of the table until you get a cross-arrow icon, then right-click and select **Cut**.

Adding an Image to a Requirement Description

To add an image to a requirement description

1. Select the requirement in which to add an image.
2. Place your cursor where to insert the image and click **Insert Image**. The **Open** dialog box displays.
3. Enter the name of the file or navigate to locate the pictureto insert.
4. When you have found the file, click **Open**. The image is inserted.
5. Click **Save** to save the change, or **Cancel Save** to cancel the change.

Assigning an Owner to a Requirement

The default Owner is the user who created the requirement. You may assign the requirement to someone else, but you may not be able to modify it again after it is saved if you do not have security privileges to do so.

To edit requirement status

1. Select the requirement with the owner to change.
2. Click the **Details** tab. The tab displays.
3. From the **Owner** drop-down box, select an owner.
4. Click **Save** to save the changes, or **Cancel Changes**to cancel the changes.

Assigning Responsible Users

Users that are accountable for the completion of a requirement are assigned responsibility for that requirement. Typically, several individuals are assigned to each requirement. For example, the business analyst who created the requirement is assigned, as well as a developer, tester and manager. When requirements are changed, the users assigned to them are notified in order to keep development on track.

To assign responsible users to a requirement

1. Select the requirement.
2. Click the **Responsibilities** tab. The tab displays project groups and member selection boxes.
3. Click the plus (+) or minus (-) sign to the left of a group/member name to expand or collapse a list. A gray check box next to a group indicates that at least one member of that group has been selected.
4. To select all members of a group, select the check box to the left of that group. To select only certain members of a group, select the check box to the left of each member you want to assign.
5. Click **Save** to save the changes, or **Cancel Changes**to cancel the changes.
6. To unassign a user, select the check box next to the user's name.

Choosing a CaliberRM Baseline

To choose a CaliberRM baseline

1. Click the **Baseline** drop-down arrow.
2. Select the baseline from the list.

Choosing a CaliberRM Project

To choose a CaliberRM project

1. Click the **Project** drop-down arrow.
2. Select a project from the list.

Creating a CaliberRM Requirement

To create a CaliberRM requirement

1 Click one of the following buttons to determine the placement of the new requirement:

- **Create Requirement**: Creates a new requirement as a child of the currently selected requirement.
- **Insert Above**: Creates a new requirement at the same hierarchical level and before the currently selected requirement.
- **Insert Below**: Creates a new requirement at the same hierarchical level and after the currently selected requirement.

A new requirement is created in the tree.

2 On each tab, enter information about the requirement.

3 After you have entered all requirement information, click **Save** to save the requirement or **Cancel Save** to cancel the creation.

Creating CaliberRM Traces

Changing an object, whether it is a requirement, a test step, or a section of source code, can potentially require changes in other elements of the project. Requirement traceability is supported to allow you to see relationships between requirements and other related development and testing information. Linking related objects together helps to ensure that changes are implemented correctly at all levels.

To create a CaliberRM trace

1. Open the application to trace to or from a CaliberRM requirement.
2. Select **View** ▸ **CaliberRM Requirements**.
3. Log on to CaliberRM.
4. Select the requirement to trace to or from.
5. Click the file that contains the code to trace.
6. Click the **Code** tab.
7. Go to the line to trace.
8. Right-click and select **Requirements** ▸ **Trace To** or **Trace From**.

   The trace appears on the **Traceability** tab for the requirement.

Deleting a CaliberRM Requirement

To delete a CaliberRM Requirement

1. Select the requirement to delete.
2. Click **Delete Requirement**

   The **Confirmation** dialog box displays.
3. Click **Yes** to delete the requirement, or **No** to cancel the deletion.

Warning: You cannot recover a requirement once it is deleted.

Displaying Requirement Numbers

Each requirement within a project has two different numbers associated with it. One is the hierarchical number, which is determined by the requirement's placement within the project tree or hierarchy. The hierarchical number changes as requirements are added, moved or deleted. The other number associated with each requirement is its unique serial (or ID) number. The serial number generally does not change, regardless of the requirement's position within a requirement type, but it will change if you move a requirement to a different type. It is not reused if the requirement is deleted. Serial numbers are composed of the requirement type tag and a number. For example, Business Requirement types may have the tag "BR." Serial numbers for Business Requirement types may have the "BR" tag as a prefix for the serial number.

To display requirement numbers

1. To display the requirement hierarchical numbers, click **Hierarchical Numbers**. The hierarchical numbers appear in the requirement list.
2. To display the requirement serial numbers, click **Serial Numbers**. The serial numbers appear in the requirement list.

Editing a Requirement

To edit a requirement

1. Select the requirement to edit.
2. Click the tab that contains the information you want to edit.
3. Make the changes.
4. Click **Save** to save the changes, or **Cancel Save** to cancel the changes.

Editing a Requirement Description

You can edit the fonts and styles, add bullets and numbering, adjust the indentation, and choose a foreground color for requirement descriptions.

To edit a requirement description

1. Select the requirement.
2. Select the text to change.
3. To change the font:
   - Select a font style in the **Font Name** box.
   - Select the font size in the **Font Size** box.
   - Select **Bold** to make the text bold.
   - Select **Italics** to make the text italic.
   - Select **Underline** to underline the text.
4. Click **Left Justify**, **Center Justify** or **Right Justify** to change the justification.
5. Click **Numbered list** or **Bullets** to add numbers or bullets to the text.
6. Click **Foreground Color**, then select a color to select a foreground color.
7. Click **OK**.
8. Click **Save** to save your changes, or **Cancel Changes** to cancel the changes.

Editing a Requirement Name

To edit a requirement name

1. Select the requirement.
2. Click the **Details** tab. The tab is displayed.
3. Type the new name or make changes to the name procedure in the **Requirement Name** field.
4. Click **Save** to save the changes, or **Cancel Changes**to cancel the changes.

Editing Requirement Priority

To edit requirement status

1. Select the requirement with the status to change.
2. Click the **Details** tab. The tab displays.
3. From the **Priority** drop-down box, select the priority.
4. Click **Save** to save the changes, or **Cancel Changes** to cancel the changes.

Editing Requirement Status

To edit requirement status

1. Select the requirement.
2. Click the **Details** tab. The tab displays.
3. From the **Status** drop-down box, select the status you want.
4. Click **Save** to save the changes, or **Cancel Changes**to cancel the changes.

Find a Requirement by ID

To search for a requirement

1. Click **Find Requirement by ID**.
2. Enter the ID number in the **Find What** field.
3. Click **Find** to search for the requirement or **Cancel** to cancel the search.

Launching CaliberRM Estimate Professional

CaliberRM Estimate Professional allows you to generate project planning estimates.

To launch CaliberRM Estimate Professional

1. Click **CaliberRM Estimate Professional**.
2. Check the check box if you want to transfer effort data from CaliberRM.
3. Select the appropriate option button for building a hierarchy.
4. Select the appropriate option button for naming tasks.
5. Click **OK**.
6. Select a project.
7. Click **OK**.

   CaliberRM Estimate Professional is launched and calculates initial effort and schedule estimates for the entire project, based on pre-defined calibration parameters.

Logging On To CaliberRM

Before you can view or update requirements in CaliberRM, you must log on to the server that contains the data for your project.

To log on to CaliberRM

1. Select **View** ▸ **CaliberRM Requirements**.
2. Type the name of the server that hosts the project you are working on.
3. Type your user name and password for the server.
4. Click **Logon**.

   To log off from CaliberRM, click **Log Off**.

   To refresh CaliberRM server data, click **Refresh**.

   Tip: You can also press **F5** to refresh data.

Modifying CaliberRM Traceability Links

You can remove a CaliberRM traceability link, make a link suspect or go to a requirement.

To modify trace link information

1. Select a requirement.
2. Click the **Traceability** tab.
3. Right-click an object in the **Traces From** or **Trace To** window.

   The following options appear in the pop-up menu.

   - **Remove Traceability**: Deletes the trace.
   - **Make Suspect**: Makes the link a suspect link. If a link is already suspect, this option is **Clear Suspect**.
   - **Go To**: Navigates to the Traceability tab of the requirement you have selected.

4. Select an option from the list.

Moving a Requirement

To move a requirement

1. Select the requirement you want to move.
2. Drag the requirement to a new location, or click **Move Requirement Up** or **Move Requirement Down**.

Posting a New Requirement Discussion Message

Project teams can provide feedback on requirements and projects through the Discussion feature. This collaborative feature enables team members to enter and reply to comments to help define, refine and prioritize requirements.

To post a new requirement discussion message

1. Select the requirement to post a message to.
2. Click the **Discussion** tab. The tab displays.
3. Click **Post New**. The **New Message** dialog box displays.
4. Type a subject for the message.
5. Type the message text, then click **Send**. The message is posted and is displayed on the **Discussion** tab.

Refreshing Discussion Messages

Project teams can provide feedback on requirements and projects through the Discussion feature. This collaborative feature enables team members to enter and reply to comments to help define, refine and prioritize requirements.

When you select a requirement's **Discussion** tab, its discussion list contains all current messages. However, if a message is added while you are in the **Discussion** tab, you may not automatically receive that message.

To refresh a requirement discussion message

1. Select the requirement with the discussion to refresh.
2. Click the **Discussion** tab.
3. Click **Refresh**.

Replying to a Discussion Message

Project teams can provide feedback on requirements and projects through the Discussion feature. This collaborative feature enables team members to enter and reply to comments to help define, refine and prioritize requirements.

To reply to a requirement discussion message

1. Select the requirement to post a reply to.
2. Click the **Discussion** tab. The tab displays.
3. Select the message to reply to.
4. Click **Reply** . The **New Reply Message** dialog box displays.
5. Type the message text.
6. Click **Send**.

   The message is posted and is displayed on the **Discussion** tab.

Requirement History

A history record for each requirement is maintained. The history record assigns revision numbers and keeps a list of changes for each revision. All changes made to a requirement, including changes to specific attributes, the requirement description, status, priority and more are recorded in the requirement's history.

To view the history of a requirement

1 Select the requirement and click the **History** tab.

The history record is made up of two parts: the **Revisions** list and the **Changes** list. The upper window contains the Revisions list. Each entry in the revisions list contains the following fields:

- **Rev #**: When a change is made, the revision number is automatically updated. The change may cause a major or minor revision number change. An administrator determines this when attributes types are defined.
- **Date/time**: The date and time the change was made.
- **Changed by**: The userid or name of the person who made the change.
- **Comment**: If a supporting comment is entered at the time the change is saved, it is displayed here.

2 Select a change in the list to see all details about a particular change. The bottom window displays a list of changes for the revision selected in the upper window. Entries in the change list contain the following fields:

- Attribute: the changed field
- Changed from: the original data
- Changed to: the new data

Requirement Validation

When you create a requirement, it is often helpful for testers to know how to verify that the requirement is implemented properly. Therefore, you can enter in a validation procedure for each requirement if you want. The validation procedure is “free form,” meaning the procedure can be any form you want, from a paragraph to a numbered list of steps.

To define the requirement validation procedure

1. Select the requirement.
2. Click the **Validation** tab. The tab displays.
3. Type the validation procedure in the **Validation Procedure** field.
4. Click **Save** to save the changes, or **Cancel Changes** to cancel the changes.

Specifying Requirement Comment Format

To specify the format for requirement comments

1. Create an application or open an existing one.
2. Login to CaliberRM.
3. Select a requirement.
4. Click **Requirement Commenting Format**.
5. Select a format.
6. Click **Save** to save the changes, or **Cancel Changes** to cancel the changes

Updating Requirement Comments

If you have added a requirement comment to source code and another user updates that requirement, you can update the comment in your code.

To update a requirement comment

1. Open the project containing requirement comments to update.
2. Click **View** ▸ **CaliberRM Requirements**.
3. Log on to CaliberRM.
4. Click on the file that contains the requirement comment.
5. Locate the requirement comment.
6. Place your cursor in the comment and right-click.
7. Select **Requirements** ▸ **Update Requirement Comments**.

   The requirement information is updated.

Viewing a CaliberRM Project Description

To view a CaliberRM project description

1. In the **Project** list, select the project from the list.
2. Select the project in the requirements tree list.

   The description displays in the **Project Info** window.

Viewing CaliberRM Custom Tabs

Custom tabs provide you a way to customize attributes and requirement information specific to your organization.

To view CaliberRM custom tabs

1. Select the requirement with the custom tab to view.
2. Select the custom tab.

   The tab displays user-defined attributes for the requirement.
3. If necessary, make edits to the attributes.
4. Click **Save** save the changes, or **Cancel Save** to cancel the changes.

Viewing CaliberRM Requirement Type Information

You can view the name, tag and description of a CaliberRM requirement type and assigned custom tabs.

To view CaliberRM requirement type information

1. Select the desired **Requirement Type** in the requirements tree list.
   The information is displayed.
2. Select the **Custom Tabs** tab to view assigned custom tabs and attributes assigned.

Compiling and Building Procedures

Building Packages

You can create packages in Developer Studio 2006 and include them in your projects.

To create a new package

1. File ▸ New ▸ Other to display the **New Items** object gallery.
2. Depending on your type of project, select either the **Delphi Projects** node, the **Delphi for .NET Projects** node, or the **C++Builder Projects** node.
3. Double-click the **Package** icon.

   This creates a new empty package and makes an entry for it in the **Project Manager**, along with two folders: one marked **Contains** and one marked **Requires**.

   Note: If you want to add required files to the package, you must add compiled packages (.dcpil, .dll) to the **Required** folder. Add uncompiled code files (.pas, .cpp, .h) to the **Contains** folder.

4. Select the package name in the Project Manager.
5. Right-click to display the drop-down context menu and choose Add to display the **Add** dialog box.
6. Browse to locate the file or files you want to add.
7. Select one or more files, and click **Open**.
8. Click **OK**.

   This adds the selected files to the package.

9. Choose Project ▸ Build <Package Name> to build the package.

To add a package to a project

1. Choose File ▸ New ▸ Other ▸ VCL Forms Application.
2. Select the project name in the **Project Manager**.
3. Right-click to display the drop-down context menu.
4. Choose **Add**.
5. Browse to locate a package file.
6. Select the file and click **Open**.
7. Click **OK**.

   This adds the package to the project.

8. Choose Project ▸ Build <Project Name> to build the project.

To add a component package to the Tool Palette

1. Choose Components ▸ Installed .NET Components.
2. Click the **.NET VCL Components** tab.
3. Click **Add**.
4. Locate the package file you want to add to the **Tool Palette**.
5. Click **Open**.

   This displays the available components from the package.

6. Click **OK**.

The components appear in the **Tool Palette**.

Finding References

The Find References refactoring feature helps you locate any connections between a file containing a symbol you intend to rename and other files where that symbol also appears. A preview allows you to decide how you want the refactoring to operate on specific targets or on the group of references as a whole.

To create a Find References list

1. Open a project.
2. Select an identifier in the **Code Editor**.
3. Choose **Search** ▸ **Find References**.

   Note: You can also invoke Find References with the keyboard shortcut Shift+Ctrl+Enter.

4. Double-click a node in the window to go to that location in the **Code Editor**.

   Note: If you continue to perform Find References operations without clearing the results, the new results are appended in chronological order to the existing results in the window.

To clear results from the Find References window

1. Select a single reference or a node.

   Note: No matter which you select, you get the same results. The entire node will be cleared.

2. Click the **Refactor Delete** icon at the top of the **Find References** window, to delete the selected item and any item in that result set.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

To clear all results from the Find References window

1. Select any item in the window.
2. Click the **Remove All References** icon at the top of the **Find References** window.

   This action clears all results from the window.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

Linking Delphi Units Into an Application

When compiling an application that references a Delphi-produced assembly, you can link the Delphi units for that assembly into your application. The compiler will link in the binary DCUIL files, which will eliminate the need to distribute the assembly with your application.

To link in a Delphi unit

1. With your application open in the IDE, choose **Project** ▸ **Add Reference**.
2. In the **Add Reference** dialog box, select a Delphi-produced assembly DLL from the list of .NET assemblies and click the **Add Reference** button.

   If the assembly you want to link to is not in the list, use the **Browse** button to find and select it.
3. Click **OK**.

   The assembly is listed in the References node of the **Project Manager**.
4. In the **Project Manager**, right-click the assembly and choose **Link in Delphi Units**.

   The menu command is disabled if the reference is not a Delphi-produced assembly.

   In the **Object Inspector**, the corresponding Link Units property is set to **True**.
5. Choose **Project** ▸ **Compile** to compile the application.

Previewing and Applying Refactoring Operations

You can preview most refactoring operations in the **Refactoring** pane. Some refactorings occur immediately and allow no preview. You might want to use the preview feature when you first begin to perform refactoring operations. The preview shows you how the refactoring engine evaluates and applies refactoring operations to various types of symbols and other refactoring targets. Previewing is set as the default behavior. When you preview a refactoring operation, the engine gathers refactoring information in a background thread and fills in the information as the information is collected.

If you apply a refactoring operation right away, it is performed in a background thread also, but a modal dialog blocks the UI activity. If the refactoring engine encounters an error during the information gathering phase of the operation, it will not apply the refactoring operation. The engine only applies the refactoring operation if it finds no errors during the information gathering phase.

To preview a refactoring operation

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.
4. Right-click to display the context menu.
5. Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.
6. Type a new name in the **New name** text box.
7. Select the **View references before refactoring** check box.
8. Click **OK**.

   This displays a hierarchical list of the potentially refactored items, in chronological order as they were found. You can jump to each item in the Code Editor.

   Note: If you want to remove an item from the refactoring operation, select the item and click the **Delete Refactoring** icon in the toolbar.

To jump to a refactoring target from the Message Pane

1. Expand any of the nodes that appear in the **Message Pane**.
2. Click on the target refactoring operation that you would like to view in the **Code Editor**.
3. Make any changes you would like in the **Code Editor**.

   Warning: If you change an item in the **Code Editor**, the refactoring operation is prevented. You need to reapply the refactoring after making changes to any files during the process, while the **Message Pane** contains refactoring targets.

To apply refactorings

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.

4 Right-click to display the context menu.

5 Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.

6 Type a new name in the **New name** text box.

7 Click **OK**.

   As long as the **View references before refactoring** check box is not selected, the refactoring occurs immediately.

> **Warning:** If the refactoring engine encounters errors, the refactoring is not applied. The errors are displayed in the **Message Pane**.

Renaming a Symbol

You can rename symbols if the original declaration symbol is in your project, or if a project depended upon by your project contains the symbol and is in the same open project group. You can also rename error symbols.

To rename a symbol

1. Select the symbol name in the **Code Editor**.
2. Right-click to display the drop-down context menu.
3. Select **Refactoring** ▸ **Rename 'symbol type' ' symbol name'** where *symbol type* is either method, variable, or field, and *symbol name* is the actual name of the selected symbol.

   This displays the **Rename** dialog box.
4. Enter the new name in the **New Name** text box.
5. If you want to preview the changes to your project files, select the **View References Before Refactoring** check box.

> **Note:** The menu commands are context-sensitive. If you select a method, the command will read **Rename Method** *method name* where *method name* is the actual name of the method you have selected. This context-sensitivity holds true for all other object types, as well.

Setting Project Options

You can manage application and compiler options for your project. Making changes to your project only affects the current project. However, you can also save your selections as the default settings for new projects.

To change compiler options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Select **Compiler** and set your options to modify how you want your program to compile.
3. Click **OK**.

To change application options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Select **Application** and specify a title and extension for your application.
3. Click **OK**.

To change debugger options

1. Choose **Project** ▸ **Options**.

   The **Options** dialog box appears.
2. Use the **Debugger** page to pass command-line parameters to your application, specify a host executable for testing a DLL, or load an executable into the debugger.
3. Use the **Environment Block** page to indicate which environment variables are passed to your application while you are debugging it.
4. Click **OK**.

Using Build Configurations

Note: Build configurations are available in only the C++ personality.

To create and use a new build configuration

1. Create a new build configuration.
2. Change build configuration settings.
3. Activate the build configuration.

To create a new build configuration

1. Choose **Project** ▸ **Build Configurations**.

   The **Build Configurations** dialog box appears.
2. Click **New** or select an existing build configuration and click **Copy**.

   The **New Build Configuration** dialog box appears.
3. Enter the name and output directory and the new build configuration and click **OK**.

The new build configuration now appears in the **Build Configurations** drop-down list on the **Project Options** dialog box.

To change build configuration settings

1. Choose **Project** ▸ **Options**.

   The **Project Options** dialog box appears.
2. Choose the build configuration you want to work with from the **Build Configurations** drop-down list.
3. Select a page that provides options for a build tool, such as **C++ Compiler**.
4. Change settings on build tool options pages.

   Note: If you are working with a build configuration other than **All Configurations**, some options might appear in blue. Options appear in blue when they override the value set in **All Configurations**.
5. Click **OK**.

To activate a build configuration

1. Choose **Project** ▸ **Build Configurations**.

   The **Build Configurations** dialog box appears.
2. Select the build configuration you want to activate and click **Activate**.

   (active) is appended to the build configuration you selected.
3. Click **OK**.

The build configuration you activated will now be used when you build your project.

Debugging Procedures

Adding a Watch

Add a watch to track the values of program variables or expressions as you step over or trace into code. Each time program execution pauses, the debugger evaluates all the items listed on the Active tab (or ActiveWatchGroup) in the **Watch List** window and updates their displayed values.

You can organize watches into groups. When you add a watch group, a new tab is added to the **Watch List** window and all watches associated with that group are shown on that tab. When a group tab is displayed, only the watches in that group are evaluated during debugging. By grouping watches, you can also prevent out-of-scope expressions from slowing down stepping.

To add a watch

1. Choose **Run** ▸ **Add Watch** to display the **Watch Properties** dialog box.
2. In the **Expression** field, enter the expression you want to watch.

   An expression consists of constants, variables, and values contained in data structures, combined with language operators. Almost anything you can use as the right side of an assignment operator can be used as a debugging expression, except for variables not accessible from the current execution point.
3. Optionally, enter a name in the **Group Name** field to create the watch in a new group, or select a group name from the list of previously defined groups.
4. Specify other options as needed (click **Help** on the **Watch Properties** dialog for a description of the options). For example, you can request the debugger to evaluate the watch, even if doing so causes function calls, by selecting the **Allow Function Calls** option.
5. Click **OK**.

The watch is added to the **Watch List** window.

Attaching to a Running Process

You can attach to a process that is running on your computer or on a remote computer. This is useful for debugging a program that was not created with Developer Studio 2006.

To attach to a running process

1. Choose **Run** ▸ **Attach to Process** to display the **Attach to Process** dialog box.
2. Select either **Borland .NET Debugger** or **Borland Win32 Debugger** from the **Debugger** drop-down list, depending on whether you want to attach to a .NET or Win32 process.

   The list of **Running Processes** is refreshed to display the appropriate processes. For Win32 processes, you can also check **Show System Processes** to include system processes in the list.
3. If the process is running on a remote computer, enter the name the computer in the **Remote Machine** field

   Note: The remote debug server must be running on the remote computer.
4. Select a process from the list of **Running Processes**.
5. If you do not want the process to pause after you have attached to it, uncheck **Pause After Attach**.
6. Click **Attach**.

Debugging Remote Applications

Remote debugging lets you debug a Developer Studio 2006 application running on a remote computer. Once the remote debug server is running on the remote computer, you can use Developer Studio 2006 to connect to that computer and begin debugging.

Use the following set of procedures to debug an application running on a remote machine

1. Enable debugging on a machine without the full IDE installation. For details on this procedure, see

   Installing a Debugger on a Remote Machine
2. Connect the local machine to the remote machine. For details on this procedure, see

   Establishing a Connection for Remote Debugging
3. Generate program files to be copied to the remote machine. For details on this procedure, see

   Preparing Files for Remote Debugging

Debugging VCL for .NET Source Code

To debug VCL for .NET source code, you must set certain project options that are not needed when debugging other types of applications. The options are off by default and must be specifically set.

To enable options for debugging VCL for .NET source code

1. Open a VCL for .NET project.
2. Choose **Project** ▸ **Options** ▸ **Compiler**.
3. Check the **Use debug DCUILs** check box.
4. Click **OK**.
5. Select any Borland-produced assembly under **References** in the **Project Manager**.
6. Right-click the assembly and choose **Link in Delphi Units**.

   This sets the **Link Units** property to True in the **Object Inspector**.
7. Repeat the previous two steps for each Borland assembly that you want to debug.

You are now able to debug VCL for .NET source code.

Tip: You can use this procedure to debug VCL for .NET assemblies produced by a third party if the debug DCUILs for those assemblies are available.

Displaying Expanded Watch Information

When you debug an application, you can inspect the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the **Watch List** window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.

To display expanded watch information in the Watch List window

1. Set a breakpoint on a valid source line within your project.

   A breakpoint icon displays in the gutter next to the selected line.

2. Choose **Run** ▸ **Add Watch** to add a watch for an object in your application.

   The watch displays in the **Watch List** window.

3. Choose **Run** ▸ **Run** to begin running the program. If needed, use the feature of the program that will cause it to run to the breakpoint you set.

   The IDE automatically switches to the Debug layout and the program stops at the breakpoint.

4. Click the **+** next to the name of the object that you added to the watch list.

   The names and values of elements of the watched object display in the **Watch List** window.

Establishing a Connection for Remote Debugging

You must establish a TCP/IP connection between the local and remote machines in preparation for remote debugging. This connection uses multiple ports that are chosen dynamically by Windows. The remote debug server listens on one port, and a separate port is opened for each application that is being debugged. A firewall that only allows connections to the listening port will prevent the remote debugger from working.

Note: If the remote machine uses the firewall included with Windows XP service pack 2, you will receive a message asking whether Borland remote debugging service should be allowed. You must indicate that this is allowed.

Warning: The connection between Developer Studio 2006 and the remote debug server is a simple TCP/IP socket, with neither encryption nor authentication support. Therefore, the remote debug server should not be run on a computer that can be accessed over the network by untrusted clients.

To connect the local machine and the remote machine

1. Ensure that the remote debugger is installed on the remote machine.
2. Ensure that the executable files and symbol files (.tds. .rsm and .pdb) have been copied to the remote machine.
3. On the remote machine, start rmtdbg100.exe with the -listen argument.

   ```
   rmtdbg100.exe -listen
   ```

   This starts the remote debugger's listener and directs it to wait for a connection from your host machine's IDE.
4. On the local machine, choose **Run** ▸ **Attach to Process**.

   This displays the **Attach to Process** dialog.
5. Specify the host name or TCP/IP address for the remote machine, then click **Refresh**.

   A list of processes running on the remote machine is displayed. This verifies the connectivity between the local and remote machines.
6. On the local machine, choose **Run** ▸ **Load Process** ▸ **Remote**.

   This displays the **Remote** page of the **Load Process** dialog.
7. In the **Remote path** field, specify the full path for the directory on the remote machine into which you copied the executable files and symbol files. The name of the executable must be included.

   For example, if you are debugging a program1.exe, and you copy this to a directory named RemoteDebugFiles \Program1 on the remote machine, specify

   `C:\RemoteDebugFiles\Program1\program1.exe`.
8. In the **Remote host** field, specify the host name or TCP/IP address for the remote machine.
9. Click the **Load** button.

   This connects the IDE on the local machine to the debugger on the remote machine.

Once this connection is established, you can use the IDE on the local machine to debug the application as it runs on the remote machine.

Note: You cannot interact directly with the remote application through the remote debugger. For interactive debugging, you can establish a remote desktop connection.

Finding References

The Find References refactoring feature helps you locate any connections between a file containing a symbol you intend to rename and other files where that symbol also appears. A preview allows you to decide how you want the refactoring to operate on specific targets or on the group of references as a whole.

To create a Find References list

1. Open a project.
2. Select an identifier in the **Code Editor**.
3. Choose **Search** ▸ **Find References**.

   Note: You can also invoke Find References with the keyboard shortcut Shift+Ctrl+Enter.

4. Double-click a node in the window to go to that location in the **Code Editor**.

   Note: If you continue to perform Find References operations without clearing the results, the new results are appended in chronological order to the existing results in the window.

To clear results from the Find References window

1. Select a single reference or a node.

   Note: No matter which you select, you get the same results. The entire node will be cleared.

2. Click the **Refactor Delete** icon at the top of the **Find References** window, to delete the selected item and any item in that result set.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

To clear all results from the Find References window

1. Select any item in the window.
2. Click the **Remove All References** icon at the top of the **Find References** window.

   This action clears all results from the window.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

Inspecting and Changing the Value of Data Elements

The **Debug Inspector** lets you inspect data elements by automatically formatting the type of data it is displaying. The **Debug Inspector** is especially useful for examining compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in the **Debug Inspector**, you can perform a *walkthrough* of compound data objects by opening a **Debug Inspector** on a component of the compound object.

Note: The **Debug Inspector** is only available when the process is stopped in the debugger.

To inspect a data element directly from the Code Editor

1. In the **Code Editor**, place the insertion point on the data element that you want to inspect.
2. Right-click and choose **Debug** ▸ **Inspect** to display the **Debug Inspector**.

To inspect a data element from the menu

1. Choose **Run** ▸ **Inspect** to display the **Inspect** dialog box.
2. In the **Inspect** dialog box, type the expression you want to inspect.
3. Click **OK**.

   The **Debug Inspector** is displayed.

Unlike watch expressions, the scope of a data element in the **Debug Inspector** is fixed at the time you evaluate it. If you use the **Inspect** command from the **Code Editor**, the debugger uses the location of the insertion point to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements that are not within the current scope of the execution point.

If you use **Run** ▸ **Inspect**, the data element is evaluated within the scope of the execution point.

If the execution point is in the scope of the expression you are inspecting, the value appears in the **Debug Inspector**. If the execution point is outside the scope of the expression, the value is undefined and the **Debug Inspector** becomes blank.

To view members of the object you are inspecting

1. Click the **Data** tab to view strings, boolean values, and other values for such things as variable name, expression, and owner.

   Tip: If you want to see the hexadecimal representation of a string, sub-inspect the string value in the **Debug Inspector**.

2. Click the **Methods** tab to view all of the methods that are members of the object's class.

   Tip: If you want to see the return type for any method, select the method and look at the status bar of the **Debug Inspector**, where the syntax line for the method, including the return type is displayed.

3. Click the **Properties** tab to view all of the properties for the active object.
4. Click any property name to see its type displayed in the status bar of the **Debug Inspector**.
5. Click the question mark (?) icon to see the actual value for that property at this point of the execution of the application.

To change the value of a data element

1. In the **Debug Inspector**, select a data element that has an **ellipsis (…)** next to it.
   The ellipsis indicates that the data element can be modified.
2. Click the **ellipsis (…)**, or right-click the element and choose **Change**.
3. Type a new value, then click **OK**.

To inspect local variable values

1. While running in Debug mode, double-click any variable that appears in the **Local Variables** window.
   This displays the **Debug Inspector** for that local variable.

Installing a Debugger on a Remote Machine

To debug a project on a machine that does not have Developer Studio 2006 installed, you must install the remote debugger executable files. You can install these files either directly from the installation disk or by copying them from a machine that has Developer Studio 2006 installed.

To install the remote debugger

1. Use the installation disk if it is available.
2. Use files from the machine that has the IDE installed if the installation disk is not available.

To install the remote debugger from the installation disk

1. Insert the installation disk into the remote machine.
2. Choose Install Remote Debugger.
3. Follow the instructions provided by the wizard.

To install the remote debugger if the installation disk is not available

1. Create a directory on the remote machine for the installation files.
2. Locate the following files on the local machine:

   - rmtdbg100.exe
   - bccide.dll
   - bordbk100.dll
   - bordbk100N.dll
   - comp32x.dll
   - dbkpro100.dll
   - DCC100.DLL
   - DCC100IL.DLL
   - Borland.dbkasp.dll

   By default, all of these files are in C:\Program Files\Borland\BDS\4.0\Bin.
3. Copy the files from your local machine to the directory you created on the remote machine.
4. On the remote computer, register bordbk100.dll and bordbk100n.dllby running the regsvr32.exe registration utility. For example, on Windows XP, enter C:\Windows\System32\regsvr32.exe bordbk100.dll at the command prompt, then enter C:\Windows\System32\regsvr32.exe bordbk100n.dll.
5. If you are debugging an ASP.NET application, copy Borland.dbkasp.dll to the Install\GlobalAssemblyCache directory on the remote machine.

   If you are debugging an ASP.NET application, register the Borland.dbkasp.dll in the GlobalAssemblyCache using the Microsoft .NET gacutil.exe utility. For example, on Windows XP with Microsoft .NET Framework SDK, enter C:\Program Files\Microsoft.NET\SDK\v1.1\Bin\gacutil Borland.dbkasp.dll.

Modifying Variable Expressions

After you have evaluated a variable or data structure item, you can modify its value. When you modify a value through the debugger, the modification is effective for the program run only. Changes you make through the **Evaluate/ Modify** dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the **Code Editor**, then recompile your program.

To change the value of an expression

1. Choose **Run** ▸ **Evaluate/Modify**.
2. Specify the expression in the **Expression** edit box.

   To modify a component property, specify the property name, for example, `this.button1.Height` or `Self.button1.Height`.
3. Enter a value in the **New Value** edit box.

   The expression must evaluate to a result that is assignment-compatible with the variable you want to assign it to. Typically, if the assignment would cause a compile or runtime error, it is not a legal modification value.
4. Choose **Modify**.

   The new value is displayed in the **Result** box.

   You cannot undo a change to a variable after you choose **Modify**. To restore a value, however, you can enter the previous value in the **Expression** box and modify the expression again.

Note: You can change individual variables or elements of arrays and data structures, but you cannot change the contents of an entire array or data structure with a single expression.

Warning: Modifying values (especially pointer values and array indexes), can have undesirable effects because you can overwrite other variables and data structures. Use caution whenever you modify program values from the debugger.

Preparing a Project for Debugging

While most debugging options are set on by default, you can use the following procedures to review and change those options. There are both general IDE options and project specific options. The project specific options vary based on the active project type, for example, Delphi, Delphi .NET, or C#.

To activate the integrated debugger

1. Choose **Tools** ▸ **Options** ▸ **Debugger Options**.
2. Select the **Integrated Debugging** option.
3. Click **OK**.
4. Optionally review the settings on the other debugging pages.

To set debug options

1. Choose **Project** ▸ **Options**.
2. Review the debugging options on the various pages of the **Project Options** dialog box.

   In particular, review the following pages: **Compiler**, **Linker**, **Directories/Conditionals**, **Version Info**, and **Debugger**. Note that not all pages are available for all project types. For example, the **Version Info** page is only displayed for Delphi Win32 projects.
3. Click **OK**.

Preparing Files for Remote Debugging

Executable files and symbol files must be copied to the remote machine after they are compiled. You must set the correct options on your local machine in order to generate these files.

To prepare files for debugging on a remote machine

1. Open the project on your local machine.
2. Choose **Project** ▸ **Options** ▸ **Linker** and verify that the **Include remote debug symbols** option is checked.

   This directs the compiler to generate a symbol file. The following extensions are used in symbol files (for Delphi projects):

Language	Debug symbol file extension
Delphi for Win32	.rsm
Delphi for .NET	.rsm and .pdb
C++	.tds
C#	.pdb

3. Compile the project on your local machine.
4. Copy the executable files and symbol files for the project to the remote machine.
5. Choose **Run** ▸ **Load Process**
6. Specify the directory into which you copied the symbol files in the **Debug symbols search path** field.
7. Click **OK**.

Previewing and Applying Refactoring Operations

You can preview most refactoring operations in the **Refactoring** pane. Some refactorings occur immediately and allow no preview. You might want to use the preview feature when you first begin to perform refactoring operations. The preview shows you how the refactoring engine evaluates and applies refactoring operations to various types of symbols and other refactoring targets. Previewing is set as the default behavior. When you preview a refactoring operation, the engine gathers refactoring information in a background thread and fills in the information as the information is collected.

If you apply a refactoring operation right away, it is performed in a background thread also, but a modal dialog blocks the UI activity. If the refactoring engine encounters an error during the information gathering phase of the operation, it will not apply the refactoring operation. The engine only applies the refactoring operation if it finds no errors during the information gathering phase.

To preview a refactoring operation

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.
4. Right-click to display the context menu.
5. Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.
6. Type a new name in the **New name** text box.
7. Select the **View references before refactoring** check box.
8. Click **OK**.

   This displays a hierarchical list of the potentially refactored items, in chronological order as they were found. You can jump to each item in the Code Editor.

   Note: If you want to remove an item from the refactoring operation, select the item and click the **Delete Refactoring** icon in the toolbar.

To jump to a refactoring target from the Message Pane

1. Expand any of the nodes that appear in the **Message Pane**.
2. Click on the target refactoring operation that you would like to view in the **Code Editor**.
3. Make any changes you would like in the **Code Editor**.

   Warning: If you change an item in the **Code Editor**, the refactoring operation is prevented. You need to reapply the refactoring after making changes to any files during the process, while the **Message Pane** contains refactoring targets.

To apply refactorings

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.

4 Right-click to display the context menu.

5 Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.

6 Type a new name in the **New name** text box.

7 Click **OK**.

   As long as the **View references before refactoring** check box is not selected, the refactoring occurs immediately.

> **Warning:** If the refactoring engine encounters errors, the refactoring is not applied. The errors are displayed in the **Message Pane**.

Refactoring Code

Refactoring refers to the capability to make structural changes to your code without changing the functionality of the code. Code can often be made more compact, more readable, and more efficient through selective refactoring operations. Developer Studio 2006 provides a set of refactoring operations that can help you re-architect your code in the most effective and efficient manner possible.

Refactoring operations are available for Delphi, C#, and C++. However, the refactorings for C# and C++ are limited in number. You can access the refactoring commands from the **Refactoring** menu or from a right-click context menu while in the **Code Editor**.

The Undo capability is available for all refactoring operations. Some operations can be undone using the standard **Undo** (CTRL+Z) menu command, while the rename refactorings provide a specific Undo feature.

To rename a symbol

1. In the **Code Editor**, click the identifier to be renamed.

   The identifier can be a method, variable, field, class, record, struct, interface, type, or parameter name.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Rename**.
3. In the **Rename** dialog box, enter the new identifier in the **New Name** field.
4. Leave **View references before refactoring** checked. If this option is unchecked, the refactoring is applied immediately, without a preview of the changes.
5. Click **OK**.

   The **Refactorings** dialog box displays every occurrence of the identifier to be changed.
6. Review the proposed changes in the **Refactorings** dialog box and use the **Refactor** button at the top of the dialog box to perform all of the refactorings listed. Use the **Remove Refactoring** button to remove the selected refactoring from the dialog box.

To declare a variable

1. In the **Code Editor**, click anywhere in a variable name that has not yet been declared.

   Note: Any undeclared variable will be highlighted with a red wavy underline by Error Insight.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Declare Variable**.

   If the variable has already been declared in the same scope, the command is not available.
3. Fill in the **Declare New Variable** dialog box as needed.
4. Click **OK**.

The variable declaration is added to the procedure, based on the values you entered in the **Declare New Variable** dialog box.

To declare a field

1. In the **Code Editor**, click anywhere in a field name that has not yet been declared.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Declare Field**.
3. Fill in the **Declare New Field** dialog box as needed.
4. Click **OK**.

The new field declaration is added to the type section of your code, based on the values you entered in the **Declare New Field** dialog box.

Note: If the new field conflicts with an existing field in the same scope, the **Refactorings** dialog box is displayed, prompting you to correct the conflict before continuing.

To create a method from a code fragment

1. In the **Code Editor**, select the code fragment to be extracted to a method.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Extract Method**.

   The **Extract Method** dialog box is displayed.
3. Enter a name for the method in the **New method name** field, or accept the suggested name.
4. Review the code in the **Sample extracted code** window.
5. Click **OK**.

Developer Studio 2006 moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the original code fragment with a call to the new method.

To convert a string constant to a resource string (for the Delphi language only)

1. In the **Code Editor**, select the quoted string to be converted to a resource string, for example, in the following code, insert the cursor into the constant Hello World:

```
procedure foo;
begin
    writeLn('Hello World');
end;
```

2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Extract Resource String**.

   Note: You can also use the Shift+Ctrl+L keyboard shortcut.

   The **Extract Resource String** dialog box is displayed.
3. Enter a name for the resource string or accept the suggested name (the `Str`, followed by the string).
4. Click **OK**.

The `resourcestring` keyword and the resource string are added to the implementation section of your code, and the original string is replaced with the new resource string name.

```
resourcestring
    strHelloWorld = 'Hello World';

procedure foo;
begin
    writeLn(StrHelloWorld);
end.
```

To find and add a namespace or unit to the uses clause

1. In the **Code Editor**, click anywhere in a the variable name whose unit you want to add to the `uses` clause (Delphi) or the namespace you want to add to the `using` clause (C#).

2 From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Find Unit**.

   The **Find Unit** dialog box displays a selection list of applicable Delphi units.

   Note: If you are coding in C#, the dialog box is called the **Use Namespace** dialog box.

3 Select the unit or namespace that you want to add to the `uses` or `using` clause in the current scope.

   You can select as many units or namespaces as you want.

4 If you are coding in Delphi, choose where to insert the reference, either in the interface section or in the implementation section.

   Note: This choice is not relevant for C# and so the selection is not available when refactoring C# code.

5 Click **OK**.

The `uses` or `using` clause is updated with the selected units or namespaces.

Renaming a Symbol

You can rename symbols if the original declaration symbol is in your project, or if a project depended upon by your project contains the symbol and is in the same open project group. You can also rename error symbols.

To rename a symbol

1. Select the symbol name in the **Code Editor**.
2. Right-click to display the drop-down context menu.
3. Select **Refactoring** ▸ **Rename 'symbol type' ' symbol name'** where *symbol type* is either method, variable, or field, and *symbol name* is the actual name of the selected symbol.

   This displays the **Rename** dialog box.
4. Enter the new name in the **New Name** text box.
5. If you want to preview the changes to your project files, select the **View References Before Refactoring** check box.

> **Note:** The menu commands are context-sensitive. If you select a method, the command will read **Rename Method** *method name* where *method name* is the actual name of the method you have selected. This context-sensitivity holds true for all other object types, as well.

Resolving Internal Errors

The error message, Internal Error: X1234 indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process.

Tip: Internal error numbers indicate the file and line number in the compiler where the error occurred. This information may help Technical Support services track down the problem. Be sure to record this information and include it with your internal error description.

To resolve an internal error

1. If the error occurs immediately after you have modified code in the editor, go back to the place where you made your changes and make a note of what was changed.
2. If you can undo or comment out the change and then recompile your application successfully, it is possible that the programming construct that you introduced exposed a problem with the compiler. If so, follow the procedure on reviewing code below.

If the problem still exists

1. Delete all of the .dcuil files associated with your project.
2. Close your project completely using **File** ▸ **Close All**.
3. Reopen your project.

   This will clear the unit cache maintained in the IDE. Alternatively, you can close the IDE and restart.
4. Another option is to try and recompile your application using the **Project** ▸ **Build** option so that the compiler will regenerate all of your dcuils.
5. If the error is still present, exit the IDE and try to compile your application using the command line version of the compiler (dccil.exe) from a command prompt. This will remove the unit caching of the IDE from the picture and could help to resolve the problem.

Review your code at the last modification point

1. If the problem still exists, go back to the place where you last made modifications to your file and review the code.

   Typically, most internal errors can be reproduced with only a few lines of code and frequently the code involves syntax or constructs that are rather unusual or unexpected. If this is the case, try modifying the code to do the same thing in a different way. For example, if you are typecasting a value, try declaring a variable of the cast type and do an assignment first.

```
begin
    if Integer(b) = 100 then...
end;
var
 a: Integer;
begin
  a := b;
  if a = 100 then...
end;
```

   Here is an example of unexpected code that you can correct to resolve the error:

```
var
    A : Integer;
begin
 { Below the second cast of A to Int64 is unnecessary; removing it can avoid the Internal
Error. }
 if Int64(Int64(A))=0 then
end;
```

2 In this case, the second cast of A to an Int64 is unnecessary and removing it corrects the error. If the problem seems to be a `while...do` loop, try using a `for...do` loop instead. Although this does not actually solve the problem, it may help you to continue work on your application.

   If this resolves the problem, it does not mean that either `while` loops or `for` loops are broken but more likely it means that the manner in which you wrote your code was unexpected.

3 Once you have identified the problem, we ask that you create the smallest possible test case that still reproduces the error and submit it to Borland.

Other techniques for resolving internal errors

1 If error seems to be on code contained within a `while...do` loop try using a `for...do` loop instead or vice versa.

2 If it uses a nested function or procedure (a procedure/function contained within a procedure/function) try unnesting them.

3 If it occurs on a typecast look for alternatives to typecasting like using a local variable of the type you need.

4 If the problem occurs within a with statement try removing the with statement altogether.

5 Try turning off compiler optimizations under **Project Options** ▸ **Compiler**.

When all else fails

1 Typically, there are many different ways to write any single piece of code. You can try and resolve an internal error by changing the code. While this may not be the best solution, it may help you to continue to work on your application. If this resolves the problem, it does not mean that either `while` loops or `for` loops are broken but perhaps that the manner in which you have written your code was unexpected and therefore resulted in an error.

2 If you've tried your code on the latest release of the compiler and it is still reproducible, create the smallest possible test case that will still reproduce the error and submit it to Borland. If it is not reproducible on the latest version, it is likely that the problem has already been fixed.

Configuring the IDE to avoid internal errors

1 Create a single directory where all of your .dcpil files (precompiled package files) are placed.

   For example, create a directory called C:\DCPIL and under **Tools Environment Options** select the **Library** tab and set the DCPIL output directory to C:\DCPIL. This setting will help ensure that the .dcpil files the compiler generates are always up-to-date. This is useful when you move a package from one directory to another. You can create a .dcuil directory on a per-project basis using **Project Options** ▸ **Directories/Conditionals** ▸ **Unit** output directory.

2 The key is to use the most up-to-date versions of your .dcuil and .dcpil files. Otherwise, you may encounter internal errors that are easily avoidable.

Setting and Modifying Source Breakpoints

Breakpoints pause program execution at a certain location or when a particular condition occurs. You can set breakpoints in the **Code Editor** before and during a debugging session. During a debugging session, any line of code that is eligible for a breakpoint is marked with a blue dot in the left gutter of the **Code Editor**.

To set a breakpoint

1. Click the left gutter of the **Code Editor** next to the line of code where you want to pause execution.
2. Choose **Run** ▸ **Add Breakpoint** ▸ **Source Breakpoint** to display the **Add Source Breakpoint** dialog box.

   Tip: To widen the **Code Editor** gutter, choose **Tools** ▸ **Options** ▸ **Editor Options** ▸ **Display** and increase the **Gutter width** option.

3. Fill in the appropriate values and click **OK**.

The following icons are used to represent breakpoints in the **Code Editor** gutter.

Icon	Description
	The breakpoint is valid and enabled. The debugger is inactive.
	The breakpoint is valid and enabled. The debugger is active.
	The breakpoint is invalid and enabled. The breakpoint is set at an invalid location, such as a comment, a blank line, or invalid declaration.
	The breakpoint is valid and disabled. The debugger is inactive.
	The breakpoint is valid and disabled. The debugger is active.
	The breakpoint is invalid and disabled. The breakpoint is set at an invalid location.

Breakpoints are displayed in the **Breakpoint List** window.

To modify a breakpoint

1. Right-click the breakpoint icon and choose **Breakpoint Properties**.
2. Set the options in the **Source Breakpoint Properties** dialog box to modify the breakpoint.

   For example, you can set a condition, create a breakpoint group, or determine what action occurs when execution reaches the breakpoint.

3. Click **Help** for more information about the options on the dialog box.
4. Click **OK**.

To create a breakpoint group

1. Right-click the breakpoint icon and choose **Breakpoint Properties**.
2. Enter a group name in the **Group** field, or select a name from the drop down list box to add the breakpoint to an existing group.
3. Click **OK**.

To enable or disable a breakpoint or breakpoint group

1. Right-click the breakpoint icon in the **Code Editor** or in the **Breakpoint List** window and choose **Enabled** to toggle between enabled and disabled.
2. To enable or disable all breakpoints, right-click a blank area (not on a breakpoint) in the **Breakpoint List** window and choose **Enable All** or **Disable All**.
3. To enable or disable a breakpoint group, right-click a blank area (not on a breakpoint) in the **Breakpoint List** window and choose **Enable Group** or **Disable Group**.

Tip: Press the Ctrl key while clicking a breakpoint in the **Code Editor** to toggle between enabled and disabled.

Disabling a breakpoint or breakpoint group prevents it from pausing execution, but retains the breakpoint settings, so that you can enable it later.

To create a conditional breakpoint

1. Choose **Run** ▸ **Add Breakpoint** ▸ **Source Breakpoint** to display the **Add Source Breakpoint** dialog box.
2. In the **Line number** field, enter the line in the **Code Editor** where you want set the breakpoint.

   Tip: To pre-fill the **Line number** field, click a line in the **Code Editor** prior to opening the **Add Source Breakpoint** dialog box.

3. In the **Condition** field, enter a conditional expression to be evaluated each time this breakpoint is encountered during program execution.
4. Click **OK**.

Conditional breakpoints are useful when you want to see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

If the conditional expression evaluates to true (or not zero), the debugger pauses the program at the breakpoint location. If the expression evaluates to false (or zero), the debugger does not stop at the breakpoint location.

To associate actions with a breakpoint

1. Choose **Run** ▸ **Add Breakpoint** ▸ **Source Breakpoint** to display the **Add Source Breakpoint** dialog box.

   Tip: You can also right-click the breakpoint icon and choose **Breakpoint Properties** to display the **Source Breakpoint Properties** dialog box.

2. Click **Advanced** to display additional options at the bottom the dialog box.
3. Check the actions that you want to occur when the breakpoint is encountered.

   For example, you can specify an expression to be evaluated and write the result of the evaluation to the **Event Log**.
4. Click **OK**.

To change the color of the text at the execution point and breakpoints

1. Choose **Tools** ▸ **Options** ▸ **Editor Options** ▸ **Color**.
2. In the code sample window, select the appropriate language tab.

   For example, to change the breakpoint color for Developer Studio 2006 code, select the Developer Studio 2006 tab.

3 Scroll the code sample window to display the execution and breakpoint icons in the left gutter of the window.

4 Click anywhere on the execution point or breakpoint line that you want to change.

5 Use the **Foreground Color** and **Background Color** drop-down lists to change the colors associated with the selected execution point or breakpoint.

6 Click **OK**.

Setting the Search Order for Debug Symbol Tables

Symbol tables are used internally during debugging. By default, Developer Studio 2006 locates and uses all symbol tables available. However, you can control the order in which these symbol tables are searched. You can also limit the search to specific symbol tables, which can speed up the debugging process.

The extensions for symbol table files vary by personality.

- Delphi Win32, does not use external symbol files because the compiler holds the symbols tables in memory. However, if you are debugging a remote application, you must generate symbol files with the .RSM extension.
- Delphi.NET, VB.NET and C# symbol files use the .PDB extension.
- C++ symbol files use the .TDS extension. However, if debug information is contained in the PE file, external symbol tables are not used.

To set the order in which symbol tables are searched

1. Specify the general project search path.
2. Specify the global path for all projects.
3. Specify the language-specific path for the project.
4. Specify the language-specific global path.

To specify the general project search path

1. Choose **Project** ▸ **Options** ▸ **Debugger** ▸ **Symbol Tables**.
2. In the **Debug symbols search path** field, type or navigate to the path to the symbols table that you want the debugger to use.

   Note: If you want to limit the search to specific symbol tables, proceed to the next step. If you want the debugger to search all paths, click **OK** to finish specifying the general project search path.

3. Uncheck the **Load all symbols** check box.
4. Click **New**.

   The **Add Symbol Table Search Path** dialog displays.

5. Enter the name of the module you are debugging and one or more paths that contain the symbol table for that module.

   If you specify multiple paths, use a semicolon to separate them.

6. Click **OK**.

   The **Add Symbol Table Search Path** dialog closes and the module and path you added are displayed in the table.

   Note: You can use this list to specify modules and paths that the debugger is to avoid searching by using a blank path and checking the **Load symbols for unspecified modules** check box.

7. Click **OK**.

To specify the global path for all projects (for Delphi and C++ only)

1. Choose **Tools** ▸ **Options** ▸ **Debugger Options** ▸ **Borland Debuggers**.

2. In the **Debug symbols search path** field, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

To specify the language-specific path for the project

1. Choose **Project** ▸ **Options** ▸ **Directories/Conditionals** .

   The **Directories/Conditionals** page contains four fields in which you can specify a path for Win32 and .NET symbol tables. They are searched in the following order during debugging:

1. **Search path**
2. **Package output directory**
3. **DCP output directory**
4. **Output directory**

2. In each of these fields, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

To specify global paths

1. Choose **Tools** ▸ **Options** ▸ **Delphi Options** ▸ **Library (Win32 or NET)**.

   Depending on the language, the **Library** page contains two or three fields in which you can specify a path for Win32 and .NET symbol tables. They are searched in the following order during debugging:

1. **Browsing path**
2. **DCP output directory** (not used for C++)
3. **Package output directory**

2. In each of these fields, type or navigate to the path to the symbols table that you want the debugger to use.
3. Click **OK**.

Using Tooltips During Debugging

When you debug an application, you can display the values of members within a watched object whose type is a complex data object (such as a class, record, or array). These values display in the code editor window when you expand a watched object. Additionally, you can expand the elements within an object, displaying its sub-elements and their values. You can expand all levels in the object. Members are grouped by ancestor.

To expand tooltips during debugging

1. Create a new VCL for Win32 application or open an existing application.
2. Choose **Project** ▸ **Options** ▸ **Compiler** and verify that the **Use debug DCUs** option is selected.
3. Choose **Tools** ▸ **Options** ▸ **Editor Options** ▸ **Code Insight** and verify that the **Tooltip expression evaluation** option is selected.
4. Choose **Run** ▸ **Step Over**.

   Tip: Alternatively, press F8.

   This opens the **Code** page of the main source file for the project.
5. Choose **Run** ▸ **Step Over** again.

   This initializes the project.
6. Move the cursor over the Application keyword.

   This displays the tooltip in a single block.
7. Click the **+** next to the Application keyword within the tooltip.

   The tooltip expands to a scrollable box that contains each child property and its value. The **+** appears next to each property that has one or more child properties. You can expand any member to display properties and values hierarchically within the tooltip.

Deploying Applications

Building Packages

You can create packages in Developer Studio 2006 and include them in your projects.

To create a new package

1. File ▸ New ▸ Other to display the **New Items** object gallery.
2. Depending on your type of project, select either the **Delphi Projects** node, the **Delphi for .NET Projects** node, or the **C++Builder Projects** node.
3. Double-click the **Package** icon.

   This creates a new empty package and makes an entry for it in the **Project Manager**, along with two folders: one marked **Contains** and one marked **Requires**.

   Note: If you want to add required files to the package, you must add compiled packages (.dcpil, .dll) to the **Required** folder. Add uncompiled code files (.pas, .cpp, .h) to the **Contains** folder.

4. Select the package name in the Project Manager.
5. Right-click to display the drop-down context menu and choose Add to display the **Add** dialog box.
6. Browse to locate the file or files you want to add.
7. Select one or more files, and click **Open**.
8. Click **OK**.

   This adds the selected files to the package.

9. Choose Project ▸ Build <Package Name> to build the package.

To add a package to a project

1. Choose File ▸ New ▸ Other ▸ VCL Forms Application.
2. Select the project name in the **Project Manager**.
3. Right-click to display the drop-down context menu.
4. Choose **Add**.
5. Browse to locate a package file.
6. Select the file and click **Open**.
7. Click **OK**.

   This adds the package to the project.

8. Choose Project ▸ Build <Project Name> to build the project.

To add a component package to the Tool Palette

1. Choose Components ▸ Installed .NET Components.
2. Click the **.NET VCL Components** tab.
3. Click **Add**.
4. Locate the package file you want to add to the **Tool Palette**.
5. Click **Open**.

   This displays the available components from the package.

6. Click **OK**.

The components appear in the **Tool Palette**.

Linking Delphi Units Into an Application

When compiling an application that references a Delphi-produced assembly, you can link the Delphi units for that assembly into your application. The compiler will link in the binary DCUIL files, which will eliminate the need to distribute the assembly with your application.

To link in a Delphi unit

1. With your application open in the IDE, choose **Project** ▸ **Add Reference**.
2. In the **Add Reference** dialog box, select a Delphi-produced assembly DLL from the list of .NET assemblies and click the **Add Reference** button.

   If the assembly you want to link to is not in the list, use the **Browse** button to find and select it.
3. Click **OK**.

   The assembly is listed in the References node of the **Project Manager**.
4. In the **Project Manager**, right-click the assembly and choose **Link in Delphi Units**.

   The menu command is disabled if the reference is not a Delphi-produced assembly.

   In the **Object Inspector**, the corresponding Link Units property is set to **True**.
5. Choose **Project** ▸ **Compile** to compile the application.

Editing Code Procedures

Creating Code Templates

While using the **Code Editor**, you can add your favorite code constructs to the **Template Manager** to create a library of the templates you use most often.

To add a Code Template using the Menu Commands:

1. While you are working in the **Code Editor**, choose **File** ▸ **New** ▸ **Other Files** and then select the **Code Template** icon.
2. Fill in the template name, description, author, and code language attributes. Then type in the code for your template between the <![CDATA[]]> tag and the </code> tag.

   Note: The Name and Language fields in the template are required.

3. Choose the **Save** command from the **File** pull-down menu in the **Code Editor** (or type CTRL + S). Your new template now appears in the IDE tree of the **Template Manager** window. It is saved, by default, in the Documents and Settings\.....\X.X(release number)\code_templates directory.

To add a Code Template using the Template Manager window:

1. In the **Code Editor**, choose **View** ▸ **Templates**.
2. In the **Template Manager** window, click the **New** button. This will put an XML outline for a code template in the **Code Editor** main window. You can also select code in the Editor before you click the **New** button.
3. Fill in the template name, description, author, and code language attributes. Then type in the code for your template between the <![CDATA[]]> tag and the </code> tag.

   Note: The Name and Language fields in the template are required.

4. Choose the **Save** command from the **File** pull-down menu in the **Code Editor** (or type CTRL + S). Your new template now appears in the IDE tree of the **Template Manager** window. It is saved, by default, in the Documents and Settings\.....\X.X(release number)\code_templates directory.

Customizing Code Editor

Borland Developer Studio 2006 lets you customize your **Code Editor** by using the available settings to modify keystroke mappings, fonts, margin widths, colors, syntax highlighting, and indentation styles.

To customize general Code Editor options

1. Choose **Tools** ▸ **Options**.
2. Click **Editor Options**.
3. Select any of the customization options and make modifications.
4. Click **OK** to apply the modifications to the **Code Editor**.

Finding References

The Find References refactoring feature helps you locate any connections between a file containing a symbol you intend to rename and other files where that symbol also appears. A preview allows you to decide how you want the refactoring to operate on specific targets or on the group of references as a whole.

To create a Find References list

1. Open a project.
2. Select an identifier in the **Code Editor**.
3. Choose **Search** ▸ **Find References**.

   Note: You can also invoke Find References with the keyboard shortcut Shift+Ctrl+Enter.

4. Double-click a node in the window to go to that location in the **Code Editor**.

   Note: If you continue to perform Find References operations without clearing the results, the new results are appended in chronological order to the existing results in the window.

To clear results from the Find References window

1. Select a single reference or a node.

   Note: No matter which you select, you get the same results. The entire node will be cleared.

2. Click the **Refactor Delete** icon at the top of the **Find References** window, to delete the selected item and any item in that result set.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

To clear all results from the Find References window

1. Select any item in the window.
2. Click the **Remove All References** icon at the top of the **Find References** window.

   This action clears all results from the window.

Note: Deleting items from the **Find References** window does not delete them from your actual code files or your project.

Previewing and Applying Refactoring Operations

You can preview most refactoring operations in the **Refactoring** pane. Some refactorings occur immediately and allow no preview. You might want to use the preview feature when you first begin to perform refactoring operations. The preview shows you how the refactoring engine evaluates and applies refactoring operations to various types of symbols and other refactoring targets. Previewing is set as the default behavior. When you preview a refactoring operation, the engine gathers refactoring information in a background thread and fills in the information as the information is collected.

If you apply a refactoring operation right away, it is performed in a background thread also, but a modal dialog blocks the UI activity. If the refactoring engine encounters an error during the information gathering phase of the operation, it will not apply the refactoring operation. The engine only applies the refactoring operation if it finds no errors during the information gathering phase.

To preview a refactoring operation

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.
4. Right-click to display the context menu.
5. Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.
6. Type a new name in the **New name** text box.
7. Select the **View references before refactoring** check box.
8. Click **OK**.

   This displays a hierarchical list of the potentially refactored items, in chronological order as they were found. You can jump to each item in the Code Editor.

   Note: If you want to remove an item from the refactoring operation, select the item and click the **Delete Refactoring** icon in the toolbar.

To jump to a refactoring target from the Message Pane

1. Expand any of the nodes that appear in the **Message Pane**.
2. Click on the target refactoring operation that you would like to view in the **Code Editor**.
3. Make any changes you would like in the **Code Editor**.

   Warning: If you change an item in the **Code Editor**, the refactoring operation is prevented. You need to reapply the refactoring after making changes to any files during the process, while the **Message Pane** contains refactoring targets.

To apply refactorings

1. Open a project.
2. Locate a symbol name in the **Code Editor**.
3. Select the symbol name.

4 Right-click to display the context menu.

5 Select **Refactoring** ▸ **Rename 'symbol type'** where *symbol type* is one of the valid types, such as method, variable, or field.

   This displays the **Rename Symbol** dialog.

6 Type a new name in the **New name** text box.

7 Click **OK**.

   As long as the **View references before refactoring** check box is not selected, the refactoring occurs immediately.

> **Warning:** If the refactoring engine encounters errors, the refactoring is not applied. The errors are displayed in the **Message Pane**.

Recording a Keystroke Macro

You can record a series of keystrokes as a macro while editing code. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session.

To record a macro

1. In the **Code Editor**, click the record macro button at the bottom of the code window to begin recording.
2. Type the keystrokes that you want to record.
3. When you have finished typing the keystroke sequence, click the stop recording button .
4. To record another macro, repeat the previous steps.

> **Note:** Recording a macro replaces the previously recorded macro.

The macro is now available to use during the current IDE session.

To run a macro

1. In the **Code Editor**, position the cursor in the code where you want to run the macro.
2. Click the macro playback button to run the macro.
   If the button is dimmed, no macro is available.

Refactoring Code

Refactoring refers to the capability to make structural changes to your code without changing the functionality of the code. Code can often be made more compact, more readable, and more efficient through selective refactoring operations. Developer Studio 2006 provides a set of refactoring operations that can help you re-architect your code in the most effective and efficient manner possible.

Refactoring operations are available for Delphi, C#, and C++. However, the refactorings for C# and C++ are limited in number. You can access the refactoring commands from the **Refactoring** menu or from a right-click context menu while in the **Code Editor**.

The Undo capability is available for all refactoring operations. Some operations can be undone using the standard **Undo** (CTRL+Z) menu command, while the rename refactorings provide a specific Undo feature.

To rename a symbol

1. In the **Code Editor**, click the identifier to be renamed.

   The identifier can be a method, variable, field, class, record, struct, interface, type, or parameter name.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Rename**.
3. In the **Rename** dialog box, enter the new identifier in the **New Name** field.
4. Leave **View references before refactoring** checked. If this option is unchecked, the refactoring is applied immediately, without a preview of the changes.
5. Click **OK**.

   The **Refactorings** dialog box displays every occurrence of the identifier to be changed.
6. Review the proposed changes in the **Refactorings** dialog box and use the **Refactor** button at the top of the dialog box to perform all of the refactorings listed. Use the **Remove Refactoring** button to remove the selected refactoring from the dialog box.

To declare a variable

1. In the **Code Editor**, click anywhere in a variable name that has not yet been declared.

   Note: Any undeclared variable will be highlighted with a red wavy underline by Error Insight.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Declare Variable**.

   If the variable has already been declared in the same scope, the command is not available.
3. Fill in the **Declare New Variable** dialog box as needed.
4. Click **OK**.

The variable declaration is added to the procedure, based on the values you entered in the **Declare New Variable** dialog box.

To declare a field

1. In the **Code Editor**, click anywhere in a field name that has not yet been declared.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Declare Field**.
3. Fill in the **Declare New Field** dialog box as needed.
4. Click **OK**.

The new field declaration is added to the type section of your code, based on the values you entered in the **Declare New Field** dialog box.

Note: If the new field conflicts with an existing field in the same scope, the **Refactorings** dialog box is displayed, prompting you to correct the conflict before continuing.

To create a method from a code fragment

1. In the **Code Editor**, select the code fragment to be extracted to a method.
2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Extract Method**.

   The **Extract Method** dialog box is displayed.
3. Enter a name for the method in the **New method name** field, or accept the suggested name.
4. Review the code in the **Sample extracted code** window.
5. Click **OK**.

Developer Studio 2006 moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the original code fragment with a call to the new method.

To convert a string constant to a resource string (for the Delphi language only)

1. In the **Code Editor**, select the quoted string to be converted to a resource string, for example, in the following code, insert the cursor into the constant Hello World:

```
procedure foo;
begin
    writeLn('Hello World');
end;
```

2. From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Extract Resource String**.

   Note: You can also use the Shift+Ctrl+L keyboard shortcut.

   The **Extract Resource String** dialog box is displayed.
3. Enter a name for the resource string or accept the suggested name (the `Str`, followed by the string).
4. Click **OK**.

The `resourcestring` keyword and the resource string are added to the implementation section of your code, and the original string is replaced with the new resource string name.

```
resourcestring
    strHelloWorld = 'Hello World';

procedure foo;
begin
    writeLn(StrHelloWorld);
end.
```

To find and add a namespace or unit to the uses clause

1. In the **Code Editor**, click anywhere in a the variable name whose unit you want to add to the `uses` clause (Delphi) or the namespace you want to add to the `using` clause (C#).

2 From either the main menu or the **Code Editor** context menu, choose **Refactor** ▸ **Find Unit**.

   The **Find Unit** dialog box displays a selection list of applicable Delphi units.

   Note: If you are coding in C#, the dialog box is called the **Use Namespace** dialog box.

3 Select the unit or namespace that you want to add to the `uses` or `using` clause in the current scope.

   You can select as many units or namespaces as you want.

4 If you are coding in Delphi, choose where to insert the reference, either in the interface section or in the implementation section.

   Note: This choice is not relevant for C# and so the selection is not available when refactoring C# code.

5 Click **OK**.

The `uses` or `using` clause is updated with the selected units or namespaces.

Renaming a Symbol

You can rename symbols if the original declaration symbol is in your project, or if a project depended upon by your project contains the symbol and is in the same open project group. You can also rename error symbols.

To rename a symbol

1. Select the symbol name in the **Code Editor**.
2. Right-click to display the drop-down context menu.
3. Select **Refactoring** ▸ **Rename 'symbol type' ' symbol name'** where *symbol type* is either method, variable, or field, and *symbol name* is the actual name of the selected symbol.

   This displays the **Rename** dialog box.
4. Enter the new name in the **New Name** text box.
5. If you want to preview the changes to your project files, select the **View References Before Refactoring** check box.

> **Note:** The menu commands are context-sensitive. If you select a method, the command will read **Rename Method** *method name* where *method name* is the actual name of the method you have selected. This context-sensitivity holds true for all other object types, as well.

Using Bookmarks

You can mark a location in your code with a bookmark and jump directly to it from anywhere in the file. You can set up to ten bookmarks. Bookmarks are preserved when you save the file and available when you reopen the file in the **Code Editor**.

To set a bookmark

1. In the **Code Editor**, right-click the line of code where you want to set a bookmark.
   The **Code Editor** context menu is displayed.
2. Choose **Toggle Bookmarks** ▸ **Bookmark n**, where *n* is a number from 0 to 9.
   A bookmark icon is displayed in the left gutter of the **Code Editor**.

Tip: To set a bookmark using the shortcut keys, press CTRL+SHIFT and a number from 0 to 9.

To jump to a bookmark

1. In the **Code Editor**, right-click to display the context menu.
2. Choose **GoTo Bookmarks** ▸ **Bookmark n**, where *n* is a number from 0 to 9.

Tip: To jump to a bookmark using the shortcut keys, press CTRL and the number of the bookmark. For example, CTRL+1 will jump you to the line of code set at bookmark 1.

To remove a bookmark

1. In the **Code Editor**, right-click to display the context menu.
2. Choose **Toggle Bookmarks** ▸ **Bookmark n**, where *n* is the number of the bookmark you want to remove.
   The bookmark icon is removed from the left gutter of the **Code Editor**.

Tip: To remove all bookmarks from a file, choose **Clear Bookmarks**.

Using Class Completion

Class completion automates the definition of new classes by generating skeleton code for Delphi class members that you declare.

To use class completion

1 In the **Code Editor**, declare a class in the interface section of a unit.

For example, you might enter the following:

```
type TMyButton = class(TButton)
    property Size: Integer;
    procedure DoSomething;
end;
```

2 Right-click on the class declaration and choose **Complete Class at Cursor**.

Tip: You can also invoke Class Completion by placing the cursor within the class declaration and pressing CTRL+SHIFT+C.

Class Completion automatically adds the read and write specifiers to the declarations for any properties that require them, and then adds skeleton code in the implementation section for each class method.

Tip: You can also use class completion to fill in interface declarations for methods that you define in the implementation section.

After invoking class completion, the sample code above appears as follows:

```
type TMyButton = class(TButton)
    private
        FSize: Integer;
        procedure SetSize(const Value: Integer);
    published
        property Size: Integer read FSize write set_Size;
        procedure DoSomething;
end;
```

The following skeleton code is added to the implementation section:

```
{ TMyButton }

procedure TMyButton.DoSomething;
begin

end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  FSize := Value;
end;
```

If your declarations and implementations are sorted alphabetically, class completion maintains their sorted order. Otherwise, new routines are placed at the end of the implementation section of the unit and new declarations are placed in private sections at the beginning of the class declaration.

Tip: The **Finish Incomplete Properties** option on the **Tools** ▸ **Options** ▸ **Explorer** page determines whether class completion completes property declarations.

Using Code Folding

Code folding lets you collapse (hide) and expand (show) your code to make it easier to navigate and read. Developer Studio 2006 generates code that contains code folding regions, but you can add your own regions as needed.

To collapse and expand code

1. In the **Code Editor**, click the minus (-) sign to the left of a code block to collapse the code.
2. Click the plus (+) sign to expand the code block.

Tip: To turn off code folding for the current edit session, press and hold Ctrl+Shift, and then K, and then O. To collapse the nearest code block, press and hold Ctrl+Shift, and then K, and E. To expand the nearest code block, press and hold Ctrl+Shift, and then K, and U. To expand all code, press and hold Ctrl+Shift and then press K, and A.

To add a code folding region

1. In the **Code Editor**, use the following preprocessor directives to surround a block of code:

```
[Delphi]
{$region 'Optional text that appears when the code block is folded'}
.
.
.
{$endregion}
```

```
[C#]
$region Optional text that appears when the code block is folded
.
.
.
$endregion
```

   The region is marked with a minus (-) sign.
2. Click the minus sign to collapse the region.

Using Code Insight

Code Insight (sometimes referred to as Code Completion) is a set of features in the **Code Editor** and the HTML **Tag Editor** that provide code completion, display code parameter lists, and show tool tips for expressions and symbols.

The hint window list box filters out all interface method declarations that are referred to by property read or write clauses. The list box displays only properties and stand-alone methods declared in the interface type.

To enable Code Insight

1. Choose **Tools** ▸ **Options**.

   The **Options** dialog box appears.
2. Under **Editor Options**, select **Code Insight**.
3. Review and set the options and color preferences as needed.
4. Click **OK**.

To use Code completion

1. Choose **Tools** ▸ **Options**.

   The **Options** dialog box appears.
2. Select **Code Insight** and enable **Code Completion**.
3. In the **Code Editor**, type an object or class name followed by a dot (.) to display a list of types, properties, methods, and events, if you are using the Delphi or C# languages. Or, if you are using the C++ language, type the name of a variable that represents a pointer to a class instance and then press Ctrl + Space to display the properties, methods, and events available in the class.
4. Select the one appropriate for the class and press ENTER.

Code Insight Examples

1. If you're using the C++ language, type the name of a variable that represents a pointer to a class instance followed by Ctrl + Space to display the properties, methods, and events available in the class. To invoke code completion for a pointer type, the pointer must first be de-referenced.

   For example, type: `self.`
2. If you're using the C++ language, type an arrow (->) for a pointer to an object. You can also type the name of non-pointer types followed by a period (.) to see its list of inherited and virtual properties, methods, and events.

   For example, type:

   ```
   var test: TRect;

   :

   :

   begin

   test.
   ```
3. Type an assignment operator or the beginning of an assignment statement and press Ctrl + Space to display a list of possible values for the variable.
4. Type a procedure, function, or method call and press Ctrl + Space to display the method and it's list of arguments.

5 Type a record to display a list of fields. (This is the same as Step 1, but uses records instead of classes.)

To use Code parameters

1 Choose **Tools** ▸ **Options**.

The **Options** dialog box appears.

2 Select **Code Insight** and enable the **Code parameters** check box.

3 In the **Code Editor**, type a method name and an open parenthesis to display the method arguments.

To use ToolTip expression evaluation

1 Choose **Tools** ▸ **Options**.

The **Options** dialog box appears.

2 Select **Code Insight** and check the **ToolTip expression evaluation** check box.

3 On the **Code Editor**, point the mouse cursor to any variable to display its current value while your program has paused during debugging.

To use ToolTip symbol insight

1 Choose **Tools** ▸ **Options**.

The **Options** dialog box appears.

2 Select **Code Insight** and check the **ToolTip symbol insight** check box.

3 In the **Code Editor**, point the mouse cursor to any identifier to display its declaration while editing your code.

Using Code Templates

Code templates are reusable code statements that are accessible from the **Code Editor**. You can insert pre-defined code segments into your code or add your own code snippets to the **Template** window.

Note: If a template has one or more jump points that are editable, it will automatically enter **SyncEdit** mode when you are inserting it into your code. The jump points allow you to navigate between different areas of the template, using the Tab key and SHIFT+Tab keys. Pressing ESC, Enter,(or pressing theTab key) from the last jump point exits **SyncEdit** mode and puts the **Code Editor** back into regular edit mode. See the link at the end of this topic for more information about **SyncEdit**.

To insert an existing Code Template into your code:

1. In the **Code Editor**, choose **View** ▸ **Templates** .
2. Expand the tree in the **Template Manager** for the language you are using, by clicking the plus sign in front of language name.
3. Put the cursor at the place in your code where you want to add the template.
4. Choose the template you want to use in the **Template Manager** window.
5. Click the **Execute** button in the **Template Manager** window.

After you have inserted a template, you will probably need to fill in data, variables, methods, or other information that is specific to your code. You can use the **Code Completion** feature with some of the templates, as described below.

To use Code Completion with your template:

1. When you are at a jump point in your template, invoke the **Code Completion** window by pressing the Ctrl + Space keys.

There are two ways to surround your code with a template. Use the procedure below that best fits your working style.

To Surround text with a template using the mouse:

1. Select the code in the **Code Editor** that you want the template to surround.
2. Click the right mouse button and choose the **Surround** command. This will give you a choice of 'surround-able' templates.
3. Choose a template from the list.

To Surround text with a template using the Template Manager window:

1. In the **Code Editor**, choose **View** ▸ **Templates**.
2. Expand the tree in the **Template Manager** for the language you are using, by clicking the plus sign in front of language name.
3. Choose the template you want to use in the **Template Manager** window.
4. Select the code in the **Code Editor** that you want the template to surround.
5. Click the `Execute` button in the **Template Manager** window.

Using Sync Edit

The Sync Edit feature lets you simultaneously edit indentical identifiers in selected code. For example, in a procedure that contains three occurrences of `label1`, you can edit just the first occurrence and all the other occurrences will change automatically.

To use Sync Edit

1. In the **Code Editor**, select a block of code that contains indentical identifiers.
2. Click the **Sync Edit Mode** icon that appears in the left gutter.

   The first indentical identifier is highlighted and the others are outlined. The cursor is positioned on the first identifier. If the code contains multiple sets of indentical identifiers, you can press TAB to move between each identifier in the selection.
3. Begin editing the first identifier. As you change the identifier, the same change is performed automatically on the other identifiers.

   By default, the identifier is replaced. To change the identifier without replacing it, use the arrow keys before you begin typing.
4. When you have finished changing the identifiers, you can exit Sync Edit mode by clicking the **Sync Edit Mode** icon, or by pressing theEsc key.

Note: Sync Edit determines indentical identifiers by matching text strings; it does not analyze the identifiers. For example, it does not distinguish between two like-named identifiers of different types in different scopes. Therefore, Sync Edit is intended for small sections of code, such as a single method or a page of text. For changing larger sections of code, consider using refactoring.

Using the History Manager

The **History Manager** lets you view and compare versions of a file, including multiple backup versions, saved local changes, and the edit buffer of unsaved changes. If you are using the StarTeam integration with Developer Studio 2006, the **History Manager** also provides version information for your local source files.

For simplicity, the following procedures uses a small text file to introduce the functionality of the **History Manager**. However, the **History Manager** is available for most files, including source code and HTML files.

To create and display file versions in the Contents page

1. Choose **Tools** ▸ **Options** ▸ **Editor Options** page and verify that the **Create Backup Files** option is checked.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Other Files** ▸ **Text** and click **OK** to display a blank text file in the **Code Editor**.
3. On line one of the file, type `First line of text` and save the file using any name and location.
4. On line two, type `Second line of text` and save the file.
5. On line three, type `Third line of text` and save the file.

   There are now three versions of the file stored in the current directory in a hidden directory named __history.
6. Click the **History** tab, which is next to the **Code** tab.

   The revision list at the top of the **Contents** tab displays three versions of the file. The first version is named ~1~, the second is named ~2~, and the current version is named File. The source viewer at the bottom of the tab displays the source for the selected version.
7. Select the different versions to display their source in the source viewer.
8. Click the **Code** tab to return to the **Code Editor** and on line four of the file, type `Fourth line of text` but **do not** save the file.

   Your change is stored in the editor buffer, but not saved to the file.
9. Review the following toolbar and icon descriptions and then use the next procedure to compare the file versions that you just created.

Tip: To sort a column on any page of the **History Manager**, click the column heading.

The toolbar at the top of the **History Manager** contains the following buttons. Not all buttons are available on all pages of the **History Manager**.

Button	Description
	Refresh revision info updates the revision list to include unsaved changes to the file.
	Revert to previous revision makes the selected version the current version and is available on the **Contents** and **Info** pages. Reverting a prior version deletes any unsaved changes in the editor buffer.
	Synchronize scrolling synchronizes scrolling in the **Contents** and **Diff** pages and the **Code Editor**. It matches the line of text that contains the cursor with the nearest matching line of text in the other view. If there is no matching text in that region of the file, it matches line numbers.
	Go to next difference repositions the source on the **Diff** page to the next block of different code.
	Go to previous difference repositions the source on the **Diff** page to the previous block of different code.

	Follow text movement locates the same line in the source viewer when switching between views.

Tip: The toolbar button functions are also available of the right-click context menus of the **History Manager** pages.

The following icons are used to represent file versions in the revision lists.

Icon	Description
	The latest saved file version.
	A backup file version.
	The file version that is in the buffer and includes unsaved changes.
	A file version that is stored in a version control repository.
	A file version that you have checked out from a version control respository.

To compare file versions using the Diff page

1. Using the file that you created in the previous procedure, click the **History** tab.
2. Click the **Diff** tab at the bottom of the **History Manager**.

   The **Differences From** and **To** panes at the top of the page shows the file versions that you can compare. At the bottom of the page, source lines that were deleted are highlighted and marked with a minus sign (–). Lines that were added are highlighted and marked with a plus sign (+). The highlighting colors depend on the **Code Editor** colors.
3. Select the different file versions in both the **Differences From** pane and the **To** pane to see the results in source viewer.

To make a prior file version the current version

1. Using the file from the previous procedures, click the **Contents** tab.
2. Right-click the ~2~ version of the file and select Revert, or click the toolbar button.

   The **Confirm** dialog box indicates that reverting the file will lose any unsaved changes in the buffer.
3. Click **Yes** on **Confirm** dialog box.

   The ~2~ version becomes the current version.
4. Return to the **Code Editor** and save the change.

Tip: The Revert command is also available on the **Info** page.

Localization Procedures

Adding Languages to a Project

You can add languages to your project by using the **Satellite Assembly Wizard** (.NET) or **Resource DLL Wizard** (Win32). For each language that you add, the wizard generates a resource module project in your project group. Each resource module project is given an extension based on the language's locale.

To add a language to a project

1. Save and build your project.
2. With your project open in the IDE, choose **Project** ▸ **Languages** ▸ **Add**.

   Alternatively, you can choose either **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **Satellite Assembly Wizard** for a .NET application or **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Resource DLL Wizard** for a Win32 application.

   The wizard is displayed.
3. Make sure your project is selected in the list that appears in the dialog and then click **Next**.
4. Click the check box next to the languages that you want to add to your project and then click **Next**.
5. Review the directory path information that the wizard will use for the language's resource modules.

   Tip: To change the path, click the path, and then click the ellipsis (...) button to browse to a different directory.

   When you are satisfied with the path information, click **Next.**
6. If no satellite assembly for the language exists yet, **Create New** appears in the **Update Mode** column. Click **Next**.

   If a resource module exists for the language in the directory you have specified, click in the **Update Mode** column to select **Update** or **Overwrite**. Choose **Update** to keep and modify the existing satellite assembly project. Choose **Overwrite** to create a new, empty project and to delete the old project and any translations it contains. Click **Next**.
7. Review the summary of what the wizard will do and click **Finish** to create or update the resource modules for the languages you have selected.

   If the wizard asks to generate a .drcil (.NET) or .drc (Win32) file, click **Yes**. Any project that uses its own resource strings (instead of previously compiled .rc files) needs a .drcil or .drc file.

   If you are sure that no new files are needed (because your project does not introduce any resource strings of its own), select **Skip drcil files that are not found** in the final dialog. This prevents the wizard from generating, or asking to generate, files.
8. Click **Yes** to compile. Click **OK** to save your project group.

The generated projects contain untranslated copies of the resource strings in your original project. By default, the Translation Manager is displayed, enabling you to begin translating the resource files.

To remove a language from a project

1. Open your project.
2. Select **Project** ▸ **Languages** ▸ **Remove**.
3. Check the languages that you want to remove and then click **Next**.
4. Click **Finish**.

The wizard removes the selected resource module from your project file, but does not delete the assemblies, the source of the assemblies, or the directories in which they reside.

To restore a language to a project

1. Choose **Project** ▸ **Languages** ▸ **Add** to start the **Satellite Assembly Wizard** or **Resource DLL Wizard**.
2. Specify the directory path of the old resource module in the appropriate dialog.
3. In the **Update Mode** column, select **Update**.

   If a resource module already exists for the language (in the directory you have specified), click in the **Update Mode** column to select **Update** or **Overwrite**. Choose **Update** to keep and modify the existing assembly project. Choose **Overwrite** to create a new, empty project and to delete the old project and any translations it contains.
4. Click **Finish**.

Editing Resource Files in the Translation Manager

After you have added languages to your project by using the **Satellite Assembly Wizard** or **Resource DLL Wizard**, you can use the Translation Manager to view and edit your resource files. You can edit resource strings directly, add translated strings to the Translation Repository, or get strings from the Translation Repository.

To edit resource strings

1. Open a project that includes languages.
2. Choose **View** ▸ **Translation Manager** ▸ **Translation Editor**.
3. Expand the project tree view to display the resource files that you want to edit.

   Tip: Use the expand and collapse icons on the toolbar above the tree view.

4. Click the resource file you want to edit. The resource strings in the file are displayed in a grid in the right pane.
5. Click the field that you want to edit and type the new text directly in the grid, right-click the field and choose **Edit** to edit the string in a dialog box, or click the **Multi-line editor** icon on the toolbar above the grid.
6. Optionally, enter a comment in the **Comment** field.
7. Optionally, set the translation status for the string by using the drop-down list in the **Status** field.
8. Click the **Save Translation** icon on the toolbar above the grid to update the resource file.

Tip: To display the original form or translated form, click the **Show original form** and **Show translated form** icons in the toolbar above the grid.

To add a resource string to the Translation Repository

1. After editing a resource string in the Translation Manager, right-click the string that you want to add to the Translation Repository.
2. Choose **Repository** ▸ **Add strings to repository**.

   The resource string is added to the Translation Repository and can be viewed by closing the Translation Manager and choosing **Tools** ▸ **Translation Repository**.

To get a resource string from the Translation Repository

1. In the Translation Manager, click the **Workspace** tab.
2. Expand the project tree view to display the resource files that you want to edit. The .resx files are listed under the **.NET Resources** node.

   The .nfm files are listed under the **Forms** node.
3. Click the resource file you want to edit.

   The resource strings in the file are displayed in a grid in the right pane.
4. Right-click the field that you want to update and choose **Repository** ▸ **Get strings from repository**.

   If the Translation Repository contains only one translation that matches the selected source string, it copies that translation into the target language column. If the Repository contains more than one match for the selected resource, its default behavior is to retrieve the first matching translation it finds.

Tip: To change this behavior, close the Transaction Manager and choose **Tools** ▸ **Translation Tools Options**, click the **Repository** tab, and change the **Multiple Find Action** setting.

To open the resource file in a text editor

1. In the Translation Manager, click the **Project** tab.
2. Click the **Files** tab.
3. Double-click the resource file that you want to update.

   The file opens in a text editor.
4. Change the file as needed and save it.

Tip: To change the text editor used by the Translation Manager, choose **Tools** ▸ **Translation Tools Options** and change executable file specified in the **External Editor** field.

Setting the Active Language for a Project

After adding languages to your project with the **Satellite Assembly Wizard** or the **Resource DLL Wizard**, the base language module is loaded when you choose **Run** ▸ **Run**. However, you can load a different language module by setting the active language for the project.

To set the active language

1. In the IDE, recompile the resource module for the language you want to use.
2. Choose **Project** ▸ **Languages** ▸ **Set Active**.

   The **Set Active Language** wizard displays a list of the languages in the project. The base language appears in angle brackets at the top of the language list, for example, <English (United States)>.
3. Select a language from the list and click **Finish**.

Setting Up the External Translation Manager

If you do not have the Developer Studio 2006 IDE, you can use the External Translation Manager (ETM) to localize an application. To use ETM, the developer must provide you with the required ETM files and project files.

Note: The Microsoft .NET Framework must be installed on your computer before you install ETM.

To set up and register the ETM files

1 Obtain the following ETM files from the developer.

By default these files are in either the Program Files\Borland\BDS\4.0\Bin or the Windows\system32 directory on the developer's computer.

> **Note:** If the developer chose to install only the Delphi for Win32 personality of Developer Studio 2006, the files marked with an asterisk (*) will not available on the developer's computer.

```
Borland.Delphi.dll *
Borland.Globalization.dll *
Borland.ITE.dll *
Borland.ITE.FormDesigner.dll *
Borland.SCI.dll *
Borland.Vcl.dll *
Borland.VclRtl.dll *
Borland.VclX.dll *
designide90.bpl
dfm90.bpl
DotnetCoreAssemblies90.bpl *
etm.exe
IDECtrls90.bpl
itecore90.bpl
itedotnet90.bpl *
rc90.bpl
ResX90.bpl *
rtl90.bpl
vcl90.bpl
vclactnband90.bpl
vclide90.bpl
vclx90.bpl
xmlrtl90.bpl
nfmrtl90.bpl *
```

2 Create a directory, such as C:\ETM.

3 Copy the ETM files from the developer into the directory.

4 Open ETM.

From Windows Explorer, double-click etm.exe. From the command line, enter etm.exe.

5 Choose **Tools** ▸ **Options** ▸ **Packages**.

6 Click the **Add** button to display the **Open** dialog box.

7 Navigate to the directory that contains the ETM files.

Make sure that the **Files of type** filter is set to **Designtime packages (dcl*.bpl)**.

8 Select all of the designtime packages in the directory and click **OK**.

The designtime packages are registered and you can now begin using ETM.

To set up the project to be translated

1. Obtain a zipped translation kit of the project to be translated from the developer. The kit should include the following:
   - a satellite assembly or resource DLL for each language to be translated
   - the .bdsproj project file generated by using **File** ▸ **Save as** in the ETM project
   - the standalone translation repository (*.tmx) files
2. Unzip the translation kit into a directory of your choice.

Updating Resource Modules

When you add an additional resource, such as a button on a form, you must update your resource modules to reflect your changes.

To update resource modules

1 Save and build your project. If you are using the ETM, reopen the saved project.

2 Update the resource modules:

- In the IDE, choose **Project** ▸ **Languages** ▸ **Update Localized Projects**.
- In ETM, choose **Project** ▸ **Run Updaters** (or press F9) or click the **Files** tab and then click the **Run Updaters** button (F9).

3 After updating in the internal Translation Manager, rebuild each resource module project by opening the project in the IDE and choosing **Project** ▸ **Compile**.

Tip: To simplify this process, you can maintain all the projects, along with the application itself, in a single project group that can be compiled from the **Project Manager** by choosing **Project** ▸ **Compile All**.

Using the External Translation Manager

Translators who do not have the Developer Studio 2006 IDE can use the External Translation Manager (ETM) instead of the Translation Manager. The steps for using the ETM are similar to those for the internal Translation Manager.

Note: ETM must be set up and operational on your computer before using the following procedure. See in the link listed at the end of this topic for details.

To run the ETM

1. To run the ETM from the command line, enter: etm.exe [files]
   where [files] is the optional project group file or the project files.
2. To run the ETM from Windows Explorer, double-click etm.exe

To localize an application using the ETM

1. In ETM, choose**File** ▸ **Open** and open the project to be translated.
2. Click the **Workspace** tab.
3. Expand the project tree view to display the resource files that you want to edit.

   Tip: Use the expand and collapse icons on the toolbar above the tree view.

4. Click the unit file that you want to edit. The resource strings in the file are displayed in a grid in the right pane.
5. Click the field that you want to edit and do one of the following:

   - type the new text directly in the grid
   - right-click the field and choose **Edit** to edit the string in a dialog box
   - click the **Multi-line editor** icon on the toolbar above the grid

6. Optionally, enter a comment in the **Comment** field.
7. Optionally, set the translation status for the string by using the drop-down list in the **Status** field.
8. Click the **Save Translation** icon on the toolbar above the grid to update the resource file.

After you have finished the translations, you can send the translated files back to the developer to add to the project.

To remove languages from your project

1. Open your project.
2. On the **Languages** tab, uncheck the check box for the language you want to remove.
3. Click the **Files** tab and click the **Run Updaters** button.

ETM removes the selected assemblies or DLLs from your project, but it does not delete them, the source of them, or the directories they reside in.

Source Control Procedures

StarTeam: Adding Files

You can place new files in your project under version control by adding them to StarTeam. When you add a file, StarTeam logs your comments for the file and sets version control properties, such as lock status and revision label.

To add a file to StarTeam

1 Create a new file or open an existing file in Developer Studio 2006.

2 Choose **StarTeam** ▸ **Add**.

> **Note:** This menu item is available only for the active file. To add any file in the project to StarTeam, right-click the file in the **Project manager** and select **StarTeam** ▸ **Add**.

If the file has not been saved, the **Save File As** dialog box displays. When the file has been saved, the **Add Files** dialog box displays.

3 Enter a description of the changes made to the file in the **Summary Comment** text box.

This step is optional.

4 Click the **Options** tab at the bottom of the dialog box.

5 Choose a lock status for the file.

The lock status lets other team members know whether or not you are working on the files.

- Unlocked indicates you do not intend to make changes.
- Exclusive indicates you intend to make changes to these files, and prevents others from checking the files in.
- Non-Exclusive indicates you are working on the files, and may possibly make changes, but other users can alter and check in the files.

6 Choose additional options in this dialog box, if appropriate. For information about these options, see the *StarTeam Add File* topic.

7 Click **OK**.

The new file is checked into the StarTeam repository. The status of the StarTeam operation is displayed in the StarTeam Messages window.

Tip: To add all new source files in a Developer Studio 2006 project to StarTeam in a single operation, choose **StarTeam** ▸ **Commit Project**.

StarTeam: Checking In Files

When you check in a file, StarTeam creates a new revision of that file. StarTeam archives either the entire file or differences between it and the last revision.

To check in the active file

1 Choose **StarTeam** ▸ **Check In**.

The StarTeam **Check In** dialog box displays.

> **Tip:** You can also right-click the file in the **Project Manager** and choose**StarTeam** ▸ **Check In**.

2 Enter a description of the changes made to the file in the **Comment** text box.

This step is optional but recommended.

> **Tip:** To enter a different comment for each file, click the **Detail Comment** button and enter a description.

3 Click the **Options** tab at the bottom of the dialog box.

4 Choose the lock status.

By default, the lock status is set to **Keep current**, which indicates that the checkin will not change the lock status of the file.

5 Choose additional options in this dialog box, if appropriate. For information about these options, see the *StarTeam Check In* topic.

6 Click **OK**.

The file is checked into the StarTeam repository. The status of the operation is displayed in the StarTeam Messages window.

> **Tip:** If you have made changes to multiple files, you can check in all files by choosing **StarTeam** ▸ **Commit Project**.

For some types of files, Developer Studio 2006 automatically generates an associated file in the same module to store resources. For example, when you create a VCL Forms application for the .NET Framework, Developer Studio 2006 generates a unit file (such as Unit1.pas) and the associated form (Unit1.nfm) file in the same module. The associated form file is maintained by Developer Studio 2006 as you make changes to the unit file. The StarTeam integration treats these paired files specially. If you check in the unit file, StarTeam checks to see if the associated form file has been modified. If the form file has been modified, StarTeam will automatically check in both files.

StarTeam: Checking Out Files

When you check out a file, StarTeam copies the requested revision of that file to the appropriate working folder. If a copy of that file is already in the working folder, it is overwritten unless the working file appears to be more recent than the checked in revision. In that case, you are asked to confirm the check out.

Note: If file renaming or deletions made in your local project conflict with changes made by another team member in the StarTeam Client, you must manually resolve the pending renaming or deletion of files. Choose **StarTeam** ▸ **Pending Renames/Deletes** to open the **Pending Renames/Deletes** dialog box and commit any pending local file renames or deletions to the repository or cancel the pending operations.

To check out files

1 Choose **StarTeam** ▸ **Check Out**.

The StarTeam **Check Out** dialog box opens to the **File List** tab.

Tip: You can also right-click a file in the **Project Manager** and choose **StarTeam** ▸ **Check Out**.

2 Verify that the check boxes next to the files to be checked out are checked.

3 Click the **Options** tab at the bottom of the dialog box and choose additional options, if appropriate. For information about these options, see the *StarTeam Check Out* topic.

This step is optional, but recommended.

4 Click **OK**.

5 If the check out operation encounters unsynchronized changes between files you already have on your working system and those that you are checking out, resolve the conflicts that appear in the **Synchronization** dialog box.

StarTeam: Committing Projects

Committing a project saves any changes to the project and its source files, creating new revisions of these files in the repository. If files have been added to or removed from your Developer Studio 2006 project, the project in the repository will reflect these changes when you commit the project.

To commit a project

1. Choose **StarTeam** ▸ **Commit Project**.

   If files have been added to the project, the StarTeam **Commit Files** dialog box displays. If you have not added new files, the **Check In** dialog box displays. Proceed to the appropriate step.

2. Fill in the information in the **Commit Files** dialog box.

   This step is optional, but recommended. For information about options in this dialog, see the *StarTeam Add Files* topic.

3. Click **OK** to close the **Commit Files** dialog box.

4. Specify the checkin conditions in the **Check In** dialog box.

   This step is optional, but recommended. For information about options in this dialog, see the *StarTeam Check In Files* topic.

5. Click **OK**.

   Any new files are added and any modified project source files are checked into the StarTeam repository. The status of the StarTeam operation displays in the **StarTeam Messages**window.

StarTeam: Comparing File Revisions

There are several ways of comparing the contents of file revisions in the StarTeam integration for Developer Studio 2006. You can compare a working version of a file with its latest revision in the repository. You can compare any two revisions of a file in the repository. You can also compare the contents of any two files in the repository. The following procedures describe how to compare the contents of file revisions using the Visual Diff comparison utility.

Note: You can use the **Alternate Applications** dialog box to configure the integrated StarTeam Client to use a different comparison utility. To open the **Alternate Applications** dialog box, choose **StarTeam** ▸ **Options** ▸ **Personal Options** and click **Alternate Applications** on the **File** page of the **Personal Options** dialog box.

To compare the active working file with the latest revision in the repository

1 Choose **StarTeam** ▸ **Difference**.

The file comparison application opens, displaying the tip revision of the file on the left and the working version on the right. Differences between the files appear in a different color. Under some circumstances, the latest checked in version may contain changes made by other team members.

2 To return to the IDE, choose **File** ▸ **Exit**.

Tip: The StarTeam file revisions show up in the **History Manager**. The **History Manager** lets you compare any two revisions of a file, including any revision of the file in the StarTeam repository, locally saved revisions, and the current content of the file in the editor buffer. File comparison in the **History Manager** does not rely on any external file comparison tools.

To compare the contents of any two files in the repository

1 Choose **StarTeam** ▸ **View Client** to open the embedded client.

Tip: Alternatively, choose **StarTeam** ▸ **Launch Client** to open the standalone StarTeam Client.

2 Locate and select the two files you want to compare.

3 Right-click the selection, and choose **Compare Contents** from the context menu.

The file comparison application opens, displaying the tip revision of the file on the left and the working version on the right. Differences between the files appear in a different color. Under some circumstances, the latest checked in version may contain changes made by other team members.

4 To return to the IDE or the StarTeam Client, choose **File** ▸ **Exit**.

StarTeam: Configuring the Integration

In addition to the configuration tasks you can perform with the StarTeam Client, the StarTeam integration lets you manage StarTeam associations for your Developer Studio 2006 projects, and set personal preferences for StarTeam behavior. The StarTeam integration lets you manage the StarTeam connection properties for Developer Studio 2006 projects with the **Manage Associations** dialog box. Using the **Manage Associations** dialog box, you can view and modify the StarTeam Server, project, and view associated with your Developer Studio 2006 project. You can also disassociate your project from StarTeam.

If you have files in your project that are located on a path that isn't under the directory containing your Developer Studio 2006 project, you must map this non-relative path to a folder in the StarTeam repository before you can check in the files. The **Manage Non-relative Working Paths** dialog box lets you map non-relative paths to StarTeam folders in the repository. This dialog box is opened from the **Manage Associations** dialog box by clicking the **Manage Non-relative Paths** button.

StarTeam also lets you set personal options that suit your work style. The **Personal Options** dialog box can be accessed from within the IDE. Personal options apply to the currently logged-on user on a given workstation.

Note: In order to map non-relative paths for your project, your Developer Studio 2006 project must not be stored in the root folder of a view. You can use the **StarTeam Associations** dialog box to remap your local working path to a child folder.

To manage StarTeam associations for your projects

1. Choose **StarTeam** ▸ **Manage Associations**.
2. If you need to alter a StarTeam association, click **Edit**.

   This opens the **StarTeam Associations** dialog box.
3. Select a server from the **StarTeam Server** drop-down list. If the StarTeam Server you want to use does not appear on the list, click the **Servers** button to add a new server or change the properties of an existing server.

   Note: If you have not previously logged on, the **Log On** dialog box requests a user name and password when you select a server.
4. Select the StarTeam project that contains your Developer Studio 2006 project from the **Project Name** drop-down list.
5. Select an existing view from the **View path** drop-down list.

   Note: A view defines the files and folders that can be accessed for a given project.
6. Click **OK** to close the **StarTeam Associations** dialog box.

   The **StarTeam Associations** dialog box will list the StarTeam associations (server, project, view, and folder path) and indicate that your project is associated.

To manage a non-relative path

1. Choose **StarTeam** ▸ **Manage Associations**.

   This opens the **StarTeam Associations** dialog box.
2. Click the **Manage Non-relative Paths** button.

   This opens the **Manage Non-relative Working Paths** dialog box, which lets you map the non-relative local working path to a folder in the StarTeam repository.

Note: The **Manage Non-relative Working Paths** dialog box opens automatically when you attempt to check in a file with a non-relative path.

3 Click **Add**, browse to and select the local (non-relative) folder that you want to map, and click **OK**.

The **Select A StarTeam Folder** dialog box appears. This dialog box lets you select a folder to which you can map. If necessary, you can create child folders in the dialog box.

4 Select a StarTeam folder for storing files from a given non-relative path, and click **OK**.

Note: The folder for files in non-relative paths must be outside the root folder path for the Developer Studio 2006 project. For example, if your local working path for your Developer Studio 2006 project is C:\Borland Studio Projects\Project1 and it maps to the folder path BDS\Project1 in View1 in the repository, then any files in non-relative paths cannot be mapped to View1\BDS\Project1 or its child folders. Therefore, if you add a file, logo.bmp that is stored locally in C:\images, you cannot map the working path to BDS\Project1\images or any other folder beneath BDS\Project1, but you can map it to BDS\images.

5 Click **Close** to close the **Manage Non-relative Working Paths** dialog box.

6 Click **Close** to close the **StarTeam Associations** dialog box.

Once you have mapped the non-relative path, files that are part of your project and located in this local working path can be checked in to StarTeam.

To modify personal options

1 Open a project that is under StarTeam control.

2 Choose **StarTeam** ▸ **Personal Options**.

The **Personal Options** dialog box appears. The StarTeam **Personal Options** dialog box contains the following pages:

- Workspace: lets you specify confirmation requirements for version control operations, display options, and other parameters that apply to the behavior and appearance of StarTeam item in the workspace.
- StarTeamMPX Server: lets you enable support for StarTeamMPX Server for the active project and subsequently opened projects.
- File: lets you set checkout options, locking options, merging options, end-of-line options, default file status repository settings, and alternate applications for editing, merging, or comparing files.
- Change Request: lets you set system tray notification parameters and locking options for change requests.
- Requirement: lets you set system tray notification parameters and locking options for requirements.
- Task: lets you set system tray notification parameters and locking options for tasks.
- Topic: lets you set system tray notification parameters and locking options for topics.

The StarTeam **Personal Options** dialog box can also be opened in the StarTeam client. Refer to the *StarTeam User's Guide* for additional information on setting personal options.

Note: StarTeamMPX Server is a part of StarTeam Enterprise Advantage, but it can be purchased separately with StarTeam Standard and StarTeam Enterprise. For more information, refer to the *StarTeamMPX Server Administrator's Guide*.

3 After you have set your personal options, click **OK** to close the dialog box.

StarTeam: Editing the Active Process Item

Selecting an active process item is a convenience that can save you time as you add files or check them in later. The active process item becomes the default selection for a process item in the **Add Files** and **Check In** dialog boxes. Within Developer Studio 2006, you can set a process item as the active process item, using the embedded client or from within the standalone StarTeam Client.

To set the active process item

1. Choose **StarTeam** ▸ **View Client** to open the embedded client or choose **StarTeam** ▸ **Launch Client** to open the standalone StarTeam Client.

   The steps for setting the active process item are the same for the embedded client and the standalone client.
2. In the upper pane of the project view window, select the process item (change request, requirement, or task) you want to set as the active process item.
3. Right-click the process item, and choose **Set Active Process Item** from the context menu.

Note: Setting a second active process item clears the first. There is also a Clear Active Process Item command on the Change Request, Requirement, and Task menus, but you will probably never use it. You do not have to use the active process item while adding files or checking them in. The active process item becomes the default selection for a process item, but you can select another appropriate item.

To edit the active process item

1. Choose **StarTeam** ▸ **Active Process Item**
2. Alternatively, locate and double-click the active process item in either the embedded client or the standalone client.

Note: Depending on how your team has set up StarTeam, you may see a different dialog box called an alternate property editor (APE). APEs are created with StarTeam Extensions. Refer to the *StarTeam Extensions User's Guide* for more information about APEs and workflow processes.

StarTeam: Finding Files in the Repository

The StarTeam integration includes a Find command to help you quickly locate files in the StarTeam repository.

To find the active working file

1. Choose **StarTeam** ▸ **Find**.
   This opens the embedded StarTeam Client, and highlights the file that is active in the **Code Editor**.
2. Alternatively, right-click a file in the **Project Manager**, and choose **StarTeam** ▸ **Find**.
   This opens the embedded StarTeam Client, and highlights the selected file.

StarTeam: Launching the Client

The StarTeam integration for Developer Studio 2006 includes a StarTeam Client for the .NET Framework. Although most of the features and information provided by the client are available from within the IDE, you can launch the client (**StarTeam** ▸ **Launch Client**) and use it as a standalone application. The standalone client provides some additional capability for managing StarTeam projects and views, and administering user accounts and servers.

To launch and use the StarTeam Client

1. Choose **StarTeam** ▸ **Launch Client**.

   The client opens the StarTeam project associated with the active Developer Studio 2006 project, and selects the project's root folder.

2. Perform source code control operations or administrative tasks as needed.

   The StarTeam Client can be used even after the IDE has been closed.

StarTeam: Locking and Unlocking Files

File locking is a way to inform other developers that you are revising a file (exclusive lock) or thinking about revising it (non-exclusive lock). File locking can be specified when files are checked in and out, and when they are added. The following procedure describes how to use the **StarTeam** menu on the menu bar to lock the active file. Files can also be locked and unlocked using the **StarTeam** context menu in the **Project Manager**.

To lock or unlock the active working file

1. Choose **StarTeam** ▸ **Lock/Unlock**.

   The **Set My Lock Status** dialog box appears.

2. Select a lock status option:

   - Unlocked—removes your exclusive or non-exclusive lock on the file
   - Exclusive—prevents others from creating a new revision of this file except you (until you release the lock or someone breaks your lock)
   - Non-exclusive—indicates that you are working on the file and may possibly make changes to it

   Depending on your privileges regarding a selected file, you may be able to break another team member's lock on it.

3. To break a lock, check the **Break Existing Lock** check box.
4. Click **OK**.

Note: Depending on your personal options (**StarTeam** ▸ **Personal Options**), you may have unlocked files that are marked read-only. This prevents you from inadvertently making changes to files that you have not locked.

StarTeam: Merging Source Files

The StarTeam integration for Developer Studio 2006 helps you avoid merge conflicts by requiring you to update when necessary before checking in changes. If merge conflicts do occur when you attempt to merge a file, StarTeam and Developer Studio 2006 alert you to the conflict, and provide a means to reconcile the merge conflicts.

If you attempt to check in or checkout a file that is not based on the tip revision of the file, StarTeam asks if you want to merge it with the tip revision. The following procedure describes how to use the Visual Merge utility to merge file contents. File merging is not supported for the checkin operation, so you must check out a file to merge it with your working file. When the merge is completed the resulting modified file revision may be checked in.

By default, StarTeam opens the merge utility only when there are conflicts between the two revisions of the file. You can change this behavior to open the merge utility for all merge conditions on the **File** page of the **Personal Options** dialog box (**StarTeam** ▸ **Personal Options**).

Note: The StarTeam Client provided with Developer Studio 2006 does not include the Visual Merge utilitiy for merging revisions of files. This utility is available with the StarTeam Windows Client, and if you have the StarTeam Windows Client installed, the StarTeam integration will use this utility by default. Alternatively, you can use the **Alternate Applications** dialog box to configure the integrated StarTeam Client to use a different merge utility. To open the **Alternate Applications** dialog box, choose **StarTeam** ▸ **Personal Options** and box, and click the **Alternate Applications** button on the **File** page of the **Personal Options** dialog box.

To merge a file on checkout

1. Choose **StarTeam** ▸ **Check Out**.

   The StarTeam **Check Out** dialog box appears.

2. Specify any checkout conditions and advanced options, and click OK.

   If a merge is required, StarTeam displays a dialog box asking if you want to merge the file now.

3. Click **Yes** to start the merge.

   The merge application opens.

4. Resolve all conflicts and apply or remove any other changes as needed.

   Visual Merge lets you quickly search for and resolve conflicts and differences between the two file revisions.

5. When you have resolved all conflicts, choose **File** ▸ **Exit** to close Visual Merge and return to the IDE.

   StarTeam tells you whether you have resolved all conflicts and asks if you wish to save the file.

6. click **Yes** to replace your working file with the merged file.

   The file status will change from Merge to Modified. The file is now ready to check in to StarTeam.

StarTeam: Migrating Projects from the SCC Interface to the StarTeam Integration

If you have projects that you manage with the StarTeam SCC interface, you can associate these files with the StarTeam integration to take better advantage of the powerful features and functions provided by the StarTeam Client. This procedure will disassociate the project from the StarTeam SCC interface. The StarTeam revision history is retained for your project files.

Tip: You need to know the name of the StarTeam Server, project, and folder in which your project is stored to complete the migration to the StarTeam interface. You can get this information quickly by opening the project and launching the StarTeam Client through the SCC interface (**Tools** ▸ **Team** ▸ **Run Scc Application**). The title bar at the top of the StarTeam main window shows the server configuration that contains the currently displayed project view along with the the StarTeam project name and the view name.

To associate an SCC controlled project with the StarTeam integration

1. Open the project in Developer Studio 2006.
2. Choose **StarTeam** ▸ **Manage Associations**.

   This opens the **Manage Associations** dialog box, which lets you associate your Developer Studio 2006 project with a StarTeam Server, project, and folder.
3. Click the **Edit** button to re-establish a connection to the StarTeam Server.

   This opens the **StarTeam Associations** dialog box.
4. In the **StarTeam Associations** dialog box, fill in the following fields:

Field	Description
StarTeam Server	Specifies the StarTeam Server where the project is stored. Select a server from the StarTeam Server drop-down list. If the StarTeam Server you want to use does not appear on the list, click the **Servers** button to add a new server or change the properties of an existing server. If you have not previously logged on, the **Log On** dialog box requests a user name and password when you select a server.
Project Name	The name of the StarTeam Project in the repository. Select the StarTeam project that contains your Developer Studio 2006 project from the **Project Name** drop-down list.
View Path	Each StarTeam project has at least one view, and may have multiple views. A view defines the files and folders that can be accessed for a given project. Select an existing view from the View Path drop-down list.
Folder Path	By default, the folder path is set to the project's root folder. To choose a different folder, click the ellipsis (…) button, and select the directory.
Logged In User Name	This is the user name used to log on to the selected StarTeam Server. This field is not editable.
Local Working Path	This is the path to the local directory containing your project. This field should not require editing.

5. After you have made your selections, click **OK** to close the **StarTeam Associations** dialog box.

   The **Manage Associations** dialog box will list the StarTeam associations (server, project, view, and folder path) and indicate that your project is associated.
6. Click **Close** to close the **Manage Associations** dialog box.

The project is automatically committed. StarTeam changes the file extension for the StarTeam SCC interface configuration file from <projectfilename>.cdp to <projectfilename>.cdp.saved to disassociate the project from the SCC interface.

The project is now associated with the StarTeam integration.

StarTeam: Placing Projects and Project Groups

StarTeam integration in Developer Studio 2006 lets you place projects and project groups into StarTeam. This places the source files from the project into the StarTeam repository and establishes a tip revision for the files. Placing a project into a StarTeam enables version control of that project and makes the project accessible to other team members.

To place a project into StarTeam

1 Choose **StarTeam** ▸ **Place Project** or **StarTeam** ▸ **Place Group**.

> **Note:** If you have not saved your project or project group, you are required to save it before continuing. When your project or project group is saved locally, the **StarTeam Association** dialog box displays. This dialog box lets you specify the details for placing your project or project group into StarTeam.

2 Select a StarTeam server from the list in the **StarTeam Association** dialog box.

The project will be stored on the selected server.

> **Note:** If the server you want to use does not appear on the list, click the **Servers** button to add a new server or change the properties of an existing server. When you select a server, the **Log On** dialog box requests a user name and password. See your StarTeam administrator for your server logon name.

3 Select a StarTeam project from the **Project Name** list, or click **New** to create a new StarTeam project.

> **Note:** When you click **New**, the **New Project** dialog box displays. Use this dialog box to specify a project name and the default working folder. The StarTeam project name must be unique. The directory specified in the **Default Working Folder** field is used as the default target directory when the project is pulled from StarTeam.

4 Select a view from the **View path** list.

A view defines the files and folders that can be accessed for a given project. If you created a new StarTeam project, there is only one view, and it has the same name as the project. You can create additional views after the project has been placed into StarTeam

5 Click **OK**.

The **Add Files** dialog box displays.

6 Fill in the information in the **Add Files** dialog box.

This step is optional but recommended. For information about this dialog box, see the *StarTeam Add Files* topic.

7 Click **OK**.

The project source files are checked into the StarTeam repository. The status of the StarTeam operation displays in the **StarTeam Messages** window.

StarTeam: Pulling Projects and Project Groups

Pulling a project from a repository configures your connection to that project in the repository and deposits the project in your own workspace. In a team environment, it connects you to the network of users who can make changes in that project.

To pull a project or a project group

1 Choose **StarTeam** ▸ **Pull**.

The **Pull Group Or Project From StarTeam** dialog box displays.

> **Note:** If none of the StarTeam Servers in your server list match the server address of the server configuration used to check in the project or project group, you are asked if you want to indicate a specific server to use for this server address.

2 Choose the server where the project is stored from the **StarTeam Server** list.

3 Select the project to be pulled from **Project name** menu.

4 Click **Browse** next to the **Local working path** field.

> **Note:** In most cases you should not use the default value for the **Local working path** field. The default value is based on the default working folder specified by the team member who placed the project into StarTeam.

5 Navigate to a new or existing empty local directory to store the project.

This directory will become the local workspace for the project.

6 Specify additional options in the **Pull Group Or Project From StarTeam** dialog box, if appropriate.

For information about these options, see the *Pull Group or Project from StarTeam* topic.

7 Click **OK** to pull.

The project or project group is pulled from the repository. The status of the StarTeam operation is displayed in the **StarTeam Messages** window.

StarTeam: Removing Files

When you remove a file from your Developer Studio 2006 project, StarTeam removes the file from the repository when you commit your project.

To remove files from StarTeam control

1. Open the Developer Studio 2006 project containing the files you want to remove.
2. Choose **Project** ▸ **Remove From Project**

   A **Remove From Project** dialog box appears.
3. Select the file or files you want to remove and click **OK**.
4. Choose **File** ▸ **Save All** to save the project.
5. Choose **StarTeam** ▸ **Commit Project** to remove the files from the StarTeam repository.

   When another team member updates his project (**StarTeam** ▸ **Update Project**), the files will be removed from his local project.

   Note: This does not delete files from the local working path.

StarTeam: Reverting Files

Using the StarTeam integration, there are a number of options for reverting your source file to a previous revision from the repository.

Warning: Reverting to a prior revision deletes any unsaved changes in the editor buffer.

To revert a file to the latest revision in the repository

1 Choose **StarTeam** ▸ **Revert**.

This discards any changes in the editor buffer for the active file, and reverts it back to the most recent revision of the file in the repository.

2 Alternatively, you can right-click a file in the **Project Manager** and choose **StarTeam** ▸ **Revert**.

Note: The StarTeam file revisions show up in the **History Manager**. The **History Manager** lets you revert a file back to any revision of the file in the StarTeam repository. You can also revert a file back to a locally saved revision of the file.

StarTeam: Updating Projects

When you update a project, StarTeam updates your project's source files with the latest revisions from the repository. If files have been added to or removed from Developer Studio 2006, your local project will also reflect these changes. If a file is in a merge state, StarTeam asks if you want to merge the changes.

Note: If file renaming or deletions made in your local project conflict with changes made by another team member in the StarTeam Client, you must manually resolve the pending renaming or deletion of files. The **Pending Renames/Deletes** dialog box (**StarTeam** ▸ **Pending Renames/Deletes**) lets you commit any pending local file renames or deletions to the repository or cancel the pending operations.

To update a project

1. Choose **StarTeam** ▸ **Update Project**.

   StarTeam updates your project source files and the project file with the latest revisions from the repository. If files have been added to or removed from the project, the local project is updated to reflect these changes. If any files are in a merge state, StarTeam asks if you want to merge the files.

2. If you have a file in a merge state, click **Yes** to merge the changes or click **No** to leave the working file unchanged.

   If you click **Yes**, the StarTeam merge utility opens.

1. Resolve all conflicts and apply or remove any other changes as needed.
2. When you have resolved all conflicts, choose **File** ▸ **Exit** to close Visual Merge and return to the IDE.
3. StarTeam tells you whether you have resolved all conflicts and asks if you wish to save the file. click **Yes** to replace your working file with the merged file.

   If you click **No**, the local working file is not updated, and the file remains in a merge state.

Together Diagram Procedures

Adding a Conditional Block

Note: If the control structure requires a condition, you can enter the condition with the in-place editor, or you can enter it using the Condition field in the **Object Inspector**.

To add a statement block to the activation bar:

1. In the **Tool Palette**, click the Conditional Block button.
2. Click the target activation bar.

Alternatively:

1. Right-click an activation bar on a sequence diagram.
2. Choose **Add** ▸ **Conditional Block** on the context menu.

To set the type of the conditional block (if, for, and so on):

1. Open the **Object Inspector**.
2. Click the drop-down arrow for your choices.

Adding a Member to a Container

You can add members to class diagram elements (containers) by using the respective context menu for the diagram element in the **Diagram** or **Model Views** or available shortcut keys to add members to a class diagram container element.

To add a member to a container:

1. Right-click the container (class, interface, and so on).
2. Choose **Add** ▸ **(Member_type)**, where, Member_type, is defined in the table above.

   Tip: You can also use keyboard shortcuts to add fields and methods to a container allowing such members. Click CTRL+W (for fields) and CTRL+M (for methods, functions).

3. You can edit the member using the in-place editor, **Object Inspector**, or source code editor.

Result: The new member is placed in the compartment of the container in the sort order for the elements in your diagrams. You can set the sort order in the **Options** dialog window.

Tip: If a container already has members, you can right-click the existing member to create an additional member using the context menu. You can also select the member, and press INSERT.

Aligning Model Elements

You can automatically rearrange all or selected model elements on a diagram.

To align model elements on a diagram:

1 Select several nodes or inner classifiers on a diagram.

2 Right-click and choose **Alignment** ▸ **(algorithm)** on the context menu. The following algorithms are available:

- Top
- Bottom
- Rigth
- Left
- Center X
- Center Y

Annotating a Diagram

Use the following actions to annotate a diagram:

1. Draw an annotation
2. Draw an annotation link
3. Type comments

To draw an annotation:

1. In the **Diagram View**, you can:

   - Hyperlink the note to another diagram or element.
   - Edit the text when its in-place editor is active.
   - Edit the properties of a note using **Object Inspector**.
   - Add an existing note from one diagram to another diagram using a shortcut. (Select **Add** ▸ **Shortcuts** from any diagram context menu.)

   In the **Object Inspector** for the note, you can:

   - Edit the text.
   - Change the foreground and background colors.
   - Change the text-only property.

To draw an annotation link:

1. Click the Note Link button on the **Tool Palette**.
2. In the **Diagram View**, click the source element.
3. Drag the link to the destination element.
4. Drop when the second element is highlighted.

Tip: You can use the **Object Inspector** to view both the client and supplier sides of the link.

To type comments:

1. To enter comments in the source code, use the **Comment** fields (Author, Since, Version) in the **Object Inspector** for the class.
2. You can also enter source code comments directly into the code using the Editor.

Assigning an Element Stereotype

You can assign a stereotype in the diagram by using the in-place editor, or by using the **Object Inspector**.

Use the following techniques to specify a stereotype:

1. Assign a stereotype by using the in-place editor
2. Assign a stereotype by using the **Object Inspector**

To assign a stereotype by using the in-place editor:

1. Double-click the stereotype name to activate the in-place editor.
2. Enter the new name.
3. Press Enter.

To assign a stereotype by using the Object Inspector:

1. Select a class on your diagram.
2. In the **Object Inspector**, select the **Stereotype** field.
3. Click the value editor button and choose the required stereotype from the combo box. Alternatively, type the stereotype name.

Result: The stereotype name is displayed in angle brackets in the class node.

Associating a Lifeline with a Classifier

To associate a lifeline with a classifier:

1. Select a lifeline on an Interaction diagram.
2. Right-click the lifeline and select **Choose ▸ Type...** on the context menu. The **Choose represented connectable element's type** dialog box opens.
3. Choose a classifier to be associated with the lifeline from the tree of available model elements.
4. Click **OK**.

Associating a Message Link with a Method

Message links can be associated with the methods of the recipient class. The methods can be selected from the list of existing ones or can be created. This is done by two commands provided by the message context menu: **Add** and **Choose method**.

You can use the **Operation** field in the **Object Inspector** to rename the method. A dialog box appears asking if you want to create a new method or rename the old one.

Use the following techniques to associate a message link with a method (operation):

1. Create a new method for an existing message link
2. Associate an existing method with a message link
3. Unlink a method

To create a new method for an existing message link:

1. Create a message link between two objects. The recipient object must instantiate a class.
2. On the context menu of the message link, choose **Add**. The submenu provides the choice of **Method**, **Constructor** or **Destructor**.

   Note: Destructors are available for classes in C# projects only.
3. From the submenu, choose the required operation type.

Tip: If the recipient object does not instantiate a class, the Add command is not available on the context menu.

If the recipient object is associated with an interface, only methods can be associated with the message link.

Result: The new operation is created in the class of the recipient object. The message link is labeled with the operation name, according to the operation type:

If a Method is selected, the label is `Method<n> ():return_type`.

If a Constructor is selected, the label is `<Classname>()` in C# projects projects.

If a Destructor is selected, the label is `~<Classname>()`. The Destructor option is disabled in the submenu of the **Add** command.

You can use the **Operation** field in the **Object Inspector** to create a new method in the classifier. For example, in the Operation field, you can enter `method_name(parameter_types):return_type`. Entering `parameter_types` is optional. If the method does not exist in the class, a dialog opens prompting you to create it. If the method already exists in the class, the message link is automatically set for that method.

To associate an existing method with a message link:

1. Create a message link between two objects. The recipient object must instantiate a class.
2. On the context menu of the message link, select **Choose method**. The submenu displays the list of operations of the recipient class.
3. If you cannot find the required operation in the list, click **More** to reveal the next 20 methods (including inherited operations) of the recipient class.
4. Select the required operation.

Result: The associated operation is selected from the list of available methods, constructor, or destructor.

If you choose to associate a different classifier for an object that is already instantiated with a classifier, all of the message links where the **Operation** property has been set are automatically saved as text unless the method signature matches another method signature within the newly-linked classifier.

To unlink a method:

1. Select the message link.
2. On the context menu of the message link, choose **Unlink method**.

Result: An association between the message link and the operation is removed. However, the operation is preserved in the recipient class.

If you unlink a classifier from an object and that object has incoming message links where the **Operation** property is set to a method of the unlinked classifier, a dialog opens prompting you to unlink the method from the message link or save it as text. Choosing the option to save as text places the **Operation** property in quotation marks and the operation displays in red on the diagram. The intent of this feature is to help users to preserve all of the signatures of any methods that have been linked to the message links. Upon instantiating the object with a class again, you can delete the quotation marks. This will open a dialog box prompting you to create the method if it does not exist in the linked classifier. A dialog box does not open if the signature of the method matches an existing method in the classifier.

Associating a Transition or a State with an Activity

You can associate an activity (created on some UML 2.0 Activity Diagram) with a state (on entering the state, while doing the state activity, and on exiting the state), or with a transition between states.

To associate a transition with an activity:

1. Select a transition or a state on a UML 2.0 State Machine diagram.
2. Under the General node of the **Object Inspector**, click the **Effect** (for a transition) or **Do activity**, **Entry** or **Exit** (for a state) field.
3. Click the chooser button to open the **Choose Activity** dialog box.
4. In the model treeview, locate the desired activity.
5. Click **OK**.

Tip: Once a guard condition or effect are specified in the **Object Inspector**, you can further edit them in the diagram by double-clicking the expression to activate the in-place editor.

Associating an Object with a Classifier

In the sequence or collaboration diagram you can create associations between objects (located on an interaction diagram) and classifiers (located on some class diagram). Instantiated classes for an object can be selected from the model, or the classes can be created and added to the model.

Note that an object can instantiate classifiers that belong to the various source-code projects within a single project group, when such projects are referenced from the project in question.

The range of available classifiers depends on the project type:

- **Design projects**: classes, interfaces
- **C# implementation projects**: classes, interfaces, structures

To associate an object with an existing classifier:

1. Select an object.
2. On the context menu of the object, select **Choose class**.
3. The submenu displays the list of available classifiers. If you cannot find the required classifier in the list, click More to reveal the model tree view.
4. In the **Choose Type to Instantiate** dialog box that opens, select a classifier from the model and click **OK**.

Tip: Alternatively, use the **Object Inspector**. Click the Instantiates field and select the classifier from the model.

Result: The object displays the fully qualified path to the instantiated classifier.

Tip: To associate an object with a classifier from a different project, add this project as a referenced one.

To create a new classifier for an existing object:

1. Select an object.
2. On the context menu, choose **Add**.
3. From the submenu, choose the desired classifier type.

Result: A new classifier is added to the model. A shortcut for the new classifier appears on the interaction diagram in question, connected with the object by a dependency link.

To unlink an object:

1. Select an object.
2. On the context menu of the object choose **Unlink class**.

Result: The association is removed, but the classifier is preserved in the model.

To navigate between classifiers and objects:

1. Select the object on the diagram.

2 Right-click and choose **Synchronize Model View** on the context menu to move focus to this classifier in the **Model View**, or choose **Go to Class Definition** to open this classifier in the source code (for implementation projects).

To create a shortcut to a classifier on an interaction diagram:

1 On the diagram, select an object that instantiates a classifier.

2 Right-click and choose **Import class** on the context menu.

Result: A shortcut to the instantiated classifier is added to the diagram.

Branching Message Links

Branching messages that start from the same location on the lifeline.

To branch a message link with the previous one:

1. Select a message link on the sequence or collaboration diagram.
2. Right-click the message link and choose **Branching** ▸ **With previous** on the context menu.

To remove branching:

1. Select the message link to remove branching from.
2. Right-click the message link and choose **Branching** ▸ **None** on the context menu.

Browsing a Diagram with Overview Pane

To open the Overview pane:

1. Open a diagram and click the **Overview** button. The pane expands to show a thumbnail image of the current diagram.
2. Click the shaded area and drag it. This is a convenient way to scroll around the diagram.
3. Resize the **Overview** pane by clicking the upper-left corner of the pane and dragging it.
4. Close the **Overview** pane by clicking the diagram.

Changing Appearance of Compartments

You can collapse or expand compartments for the different members of class, interface, namespace, enum, and structure (C# projects only) elements. By default, the compartments for these elements are displayed on the diagram as a straight line. You can use the **Options** dialog window to set viewing preferences for compartment controls. Adding compartment controls is particularly useful when you have large container elements with content that does not need to be visible at all times.

To collapse or expand compartments:

1. Select the class (or interface) on the diagram.
2. Click the "**+**" or "**-**" in the left corner of the compartment.

To view the compartment controls:

1. Open the Options dialog window.
2. Select the **Together** ▸ **(level)** ▸ **Diagram** ▸ **Appearance** ▸ **Nodes** category.
3. In this category, edit the **Show compartments as line field**.

Changing Appearance of Interfaces

To show an interface as a circle sing the context menu:

1. Right-click the interface element in the **Diagram** or **Model View**s.
2. Choose **Show as circle**.

Tip: This menu item works as a toggle. Right-click again and choose **Show as circle** to show the interface element as a rectangle.

Note: Interfaces shown as small circles do not show their members in the **Diagram View**. Use the **Model View** to view the members.

To show an interface as a circle using the Object Inspector:

1. Select the interface element in the **Diagram** or **Model View**s.
2. Press F4 to open the **Object Inspector**.
3. Set the **Circle view** property as True.

Tip: Choose False for the **Circle view** property to show the interface element as a rectangle.

Changing Diagram Notation

Use the following techniques to change diagram notation:

1. Choose one of the two possible appearances for interfaces. Interfaces can be represented as rectangles or small circles ("lollipops").
2. In UML 2.0 projects, you can change notation of interfaces to "ball and socket".
3. Adjust appearance options, including selection between UML or language formats.

   Tip: Notation options are included in the **Diagram** ▸ **Appearance** category of Together options.

4. Use the **UML In Color** profile.
5. Use stereotypes.

Changing Type of a Link

Use the following techniques to change the type of a link:

1. Set the link type by using the Object Inspector
2. Set the link type by using the context menu

To set the link type by using the Object Inspector:

1. Choose View | **Object Inspector** if the **Object Inspector** is not open.
2. Select a link on the diagram. The properties for the link appear in the **Object Inspector**.
3. In the **Object Inspector**, select the **Type** field.
4. Click the drop-down arrow and select the appropriate property from the list. Your available choices are association, aggregation, or composition.

To set the link type by using the context menu:

1. Right-click a link on the diagram.
2. Choose **Link Type** on the context menu.

Closing a Diagram

To close a diagram:

1. Switch to the **Diagram View**.
2. Click the cross icon to close the current view.

Note: Closing a diagram in the **Diagram View** does not remove it from your project.

Converting Between UML 1.5 Sequence and Collaboration Diagrams

You can convert between sequence and collaboration diagrams. However, when you create a new diagram, you must specify that it is either a sequence diagram or a collaboration diagram.

To convert between sequence and collaboration diagrams:

1. Right-click the diagram background.
2. If the diagram is a sequence diagram, choose **Show as Collaboration** on the context menu. If the diagram is a collaboration diagram, choose **Show as Sequence**.
3. Repeat this process to switch back and forth.

After you convert from a sequence diagram to a collaboration diagram for the first time, or if you have added new objects to the sequence diagram between conversions, it is recommended that you perform a full layout on the collaboration diagram.

Copying and Pasting an Execution or Invocation Specification

Clipboard operations are supported for the execution and invocation specifications.

To copy and paste an execution or invocation specification:

1. **Cut**, **Copy**, and **Paste** commands are available on the context menu of an execution specification and invocation specification. It is possible to copy or move these elements within the same diagram or to another diagram.
2. When an execution or invocation specification is copied, it means that the entire branch of messages is copied also. Pasting the clipboard contents to a target lifeline results in changing the message numbers according to the numbering of messages in the target lifeline.
3. If you paste an invocation or execution specification to another diagram, the entire outgoing bunch of messages will be pasted also, with all the respective lifelines. If the target diagram does not contain lifelines for this execution specification, they will be created automatically.

Tip: It is also possible to move and copy message branches using the drag-and-drop technique. To move an execution or invocation specification, drag-and-drop it to the target location. To create a copy, drag-and-drop while holding the CTRL key down.

Copying and Pasting Model Elements

The move and copy operations are performed by drag-and-drop, context menu commands, or keyboard shortcut keys.

Note: You can move or copy an entire diagram. In this case, all elements addressed on this diagram are not copied, and a new diagram contains shortcuts to these elements.

To copy an element:

1. Select the element or elements to be copied.
2. Do any of the following:
   - Right-click and choose **Copy** on the context menu
   - Press CTRL+C on the keyboard
3. Do any of the following:
   - Right-click the target location and choose **Paste** on the context menu
   - Select the target location and press CTRL+V

Creating a Browse-Through Sequence

The hyperlinking feature of Together allows you to create browse-through sequences comprised of any number of use case or any other diagrams.

To create a browse-through sequence:

1. You can link entire diagrams at one level of detail to the next diagram up or down in a sequence of increasing granularity, or you can link from key use cases or actors to the next diagram. By browsing the hyperlink sequence, you can follow the relationships between the use case diagrams.
2. Together does not confine hyperlinking to such sequences, however. You can use hyperlinking to link diagrams and elements based on your requirements. For example, you can create a hierarchical browse-through sequence of use case diagrams, creating hyperlinks within the diagrams that follow a specific actor through all use cases that reference the actor.

Creating a Deferred Event

You can add a deferred event to a state element.

To create a deferred event:

1. Select the desired state or activity element on the diagram or in the **Model View**.
2. Right-click the element, and select **Add** ▸ **Deferred Event** on the context menu.

Creating a Delegation Connector

To create a delegation connector:

1. Right-click an interface and choose **New** ▸ **Delegation connector** from the context menu.
2. In the **Choose Destination** dialog box that opens, select the target interface from the Model or Favorites.
3. Click **OK**.

Creating a Diagram

When you create a new diagram, the **Diagram View** presents an empty background. You place the various model elements on the background and draw relationship links between them according to the requirements of your model.

To create a diagram:

1. In the **Model View**, right-click the target project.

   Tip: Alternatively, you can use the shortcut CTRL+SHIFT+D.

2. Select the target namespace (package) either in the **Diagram View** or in the **Model View**. If you do not select a custom namespace (package), Together adds a new diagram to the default one.
3. Choose **Add** ▸ **Other Diagram** on the context menu.
4. In the **Add New Diagram** dialog box, choose the **Diagrams** tab.
5. Select the diagram type.
6. In the **Name** field, enter a name for the new diagram.
7. Click **OK**.

Result: The new diagram opens in a new tab in the Editor Window. You can use the **Object Inspector** to view and edit the diagram properties.

To create a new diagram, use can also use the **Hyperlink** ▸ **To New diagram** command on the context menu of the **Model View** or the **Diagram View**.

You can create a new logical class diagram using the context menu of the root node for your project, or by using the context menu of a namespace element in the **Model View**. Choose either **Add** ▸ **Class Diagram** or **Add** ▸ **Other Diagram**. Choosing the latter command opens the **Add New Diagram** dialog box. When you place a class, interface, or namespace on a logical class diagram, Together generates the corresponding source code or descendent namespace in the namespace where this class diagram is located.

Creating a Guard Condition for a Transition

To create a guard condition for a transition:

1. Select a transition in the diagram.
2. Under the General node of the **Object Inspector**, click the **Guard** field.
3. Type the condition expression and apply changes.

Creating a History Element

To create a history element for a state:

1. In the target state on a state diagram, select the target region where history needs to be added.
2. Choose **Shallow History** or **Deep History** on the diagram **Tool Palette**.
3. Click the target region.

Creating a Link with Bending Points

If your diagram is densely populated, you can draw bent links between the source and target elements to avoid other elements that are in the way.

To create a link with bending points:

1. Click the link button on the **Tool Palette**.
2. Click the source element.
3. Drag the link line, clicking the diagram background each time you want to create a section of the link. Sections on a link lie between two blue bullets. The bullets display whenever you select the link on the diagram.
4. Click the destination element to terminate the link.

Tip: Once you have created a link, you can add bending points to it. Select the link on the diagram, and then drag the link to the desired position. The figure below demonstrates this technique.

Creating a Member for a State

To create a member for a state:

1 Open the **Diagram View**.

2 Right-click an existing state and choose **Add** ▸ **(member)** on the context menu.

The following members are available:

- Internal transaction
- Entry point
- Exit point
- Region

Creating a Multiple Transition

To create a multiple transition (a fork or a join):

1. Identify the states involved. If necessary, place all of the states on the diagram first and arrange as desired.
2. On the diagram **Tool Palette** choose the fork or join button.
3. Place either a horizontal or vertical fork or join on the diagram.
4. Resize as needed.
5. On the diagram **Tool Palette**, choose the transition button.
6. Draw links from the source state(s) to the fork/join node, and from the fork/join node to the target state(s).

Creating a Pin

To add an input pin, output pin, or value pin, do one of the following:

1. Right-click an action.
2. Choose **New** ▸ **Input Pin (or: Output Pin, or: Value Pin)** on the context menu.

Result: The created pin is added to the target action as a square. Note that the pins are attached to their actions, and can be only dragged along the action borders.

Alternatively:

1. Open the **Tool Palette**.
2. Choose the appropriate button, and click the target action.

Creating a Port

To create a port:

1. Choose the port icon on the **Tool Palette**.
2. Click the target class or part.
3. Create as many ports as required.

Creating a Referenced Part

To create a referenced part:

1 Open the **Diagram View**.

2 Do one of the following:

- Use the referenced part button on the diagram **Tool Palette**.
- Right-click a target container and choose **New** ▸ **Referenced part** on the context menu.
- Select a part, open the **Model View**, and check the option aggregated by reference.

Creating a Self-Transition

To create a self-transition:

1. Draw a transition from the state or activity element and drag the link away from the element.
2. Drag the link back to the element and drop it.

Alternatively:

1. Draw a transition between two activities (or states).
2. Drag the opposite end of the link line back to the desired activity (or state).

Creating a Sequence or Communication Diagram from an Interaction

To create a sequence or a communication diagram from an interaction:

1. In the **Model View**, choose an Interaction element.
2. Right-click the Interaction node and choose **Open with Sequence diagram 2.0** or **Open with Communication diagram 2.0**.

Results: If such diagram is missing, it will be created. Then this diagram opens in the **Diagram View**.

Creating a Shortcut

You can create a shortcut to a model element on the diagram background by using three methods:

- By opening **Add Shortcuts** dialog box from the **Diagram View**
- By copying and pasting a shortcut from the **Model View**
- By choosing **Add Shortcuts** on the **Model View** context menu

Use the following techniques to create a shortcut:

1. Create a shortcut by using the **Add Shortcuts** dialog window
2. Create a shortcut by using drag-and-drop
3. Create a shortcut by copying and pasting
4. Create a shortcut by using the **Model View** context menu

To create a shortcut by using the Add Shortcuts dialog window:

1. Right-click the diagram background.
2. Choose **Add** ▸ **Shortcuts** on the context menu.

   Tip: You can also use CTRL+SHIFT+M to open the **Edit shortcuts** dialog window.

3. In the **Edit shortcuts** dialog window, choose the required element from the tree view of available contents.
4. Click **Add** to place the selected element to the list of the existing or ready to add elements.
5. When the list of ready to add elements is complete, click **OK**.

To create a shortcut by using drag-and-drop:

1. Select the element in the **Model View**.
2. Drag-and-drop the element onto the diagram.

To create a shortcut by copying and pasting:

1. In the **Model View**, right-click the element to be added to the current diagram as a reference.
2. Choose **Copy** on the context menu.
3. Right-click the target diagram and choose **Paste Shortcut** on the context menu.

   Tip: You can also copy an element from one diagram and paste it in another diagram as a shortcut.

To create a shortcut by using the Model View context menu:

1. Open the diagram where the shortcut will be added.
2. In the **Model View**, select the element to be added to the current diagram as a shortcut.

3 Right-click the element in the **Model View**, and choose **Add as Shortcut** on the context menu.

Creating a Simple Link

In a design project, you can create a link to another node, or a shortcut of an element of the same or another design project (these projects must be of the same UML version).

In an implementation project, you can create a link to another node or a shortcut of an element of the same project.

To create a simple link between two nodes:

1. On the diagram **Tool Palette**, click the button for the type of link you want to draw in the diagram. The button stays down.
2. Click the source element.
3. Drag to the destination element and drop when the second element is highlighted.

Creating a Single Model Element

To create a single model element:

1. Open a target diagram in the **Diagram View**.
2. Choose **Tool Palette** from the View menu.
3. Choose the **UML [diagram type]** tab in the **Tool Palette** to view available model elements.
4. On the **Tool Palette**, click the icon for the element you want to place on the diagram. The button stays down.

   Tip: Icons are identified with labels.

5. Click the diagram background in the place where you want to create the new element. This creates the new element and activates the in-place editor for its name.

Tip: Alternatively, you can right-click the diagram background and choose **Add** on the context menu. The submenu displays all of the basic elements that can be added to the diagram, and the **Shortcuts** command.

Creating a State

To create a state:

1 Using the **Tool Palette** buttons: On the diagram **Tool Palette**, choose to create a state node. Click an appropriate place on your diagram.

Alternatively:

Using the context menu of the diagram: Right-click the diagram background. Select **Add** ▸ **State** on the context menu.

> **Note:** You can place a state inside of the existing state. It is possible to hide individual states. For example, you might want to hide the content of composite states for better understanding of the whole diagram.

2 When a new state is placed on a diagram, you can use the **Object Inspector** to adjust its properties, including:

- Configure standard properties of the element.
- In the State Invariant field, select the language of the expression from the Language list box. The possible options are OCL and plain text.
- In the Properties page, configure the behavior of the state by setting or viewing the following additional properties:

Field	Description
Composite	Set to True if there is one or more regions in this state (not editable)
Orthogonal	Set to True if there are two or more regions in this state (not editable)
Simple	Set to True if there are no regions in this state (not editable)
Do activity	Specify the activity to be performed during execution of the current state by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project
Entry	Specify the activity to be performed when the current state starts executing by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project
Exit	Specify the activity to be performed when the current state finishes executing by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project

In the edit field below the list box enter the OCL expression for this state.

Creating a State Invariant

To create a state invariant as an OCL comment:

1. On the diagram **Tool Palette**, choose the state invariant button.
2. Click the target lifeline or execution specification.

   Tip: Alternatively, use the **Add** ▸ **State invariant** command on the context menu of a lifeline or an execution specification.

3. In the **Object Inspector** of the state invariant, select the General node.
4. In the Invariant kind field, choose OCL expression from the drop-down list. The shape of the state invariant diagram element changes to braces.
5. In the OCL invariant node that adds to the Property Browser, select the language of the comment from the Language drop-down list. The possible options are OCL and plain text.
6. Type the text and apply changes.

To connect a state invariant to a state:

1. On the diagram **Tool Palette**, choose the state invariant button.
2. Click the target lifeline or execution specification.
3. In the **Object Inspector** of the state invariant, select the General node.
4. In the Invariant kind field, choose States/Regions from the drop-down list.
5. In the States/Regions field, click the chooser button.
6. In the **Choose States and/or Regions** dialog box, select the desired states and/or regions from the model, using the Add button.
7. Click OK when ready.

   Tip: Alternatively, type the state or region name. If the state or region belongs to a different package, specify its fully-qualified name.

Creating an Activity for a State

To create an activity for a state:

1. Open the **Diagram View**.
2. Right—click a state and choose **Add** ▸ **Activity** on the context menu.

Result: A new activity is created inside of a state.

Creating an Association Class

To create an association class:

1. On the diagram **Tool Palette**, select the association class button.
2. Click the diagram background. This adds a regular class icon for the association class, connected with the diamond icon.
3. Create participant classes.
4. Using the association end button, connect the n-ary association with the participant classes.

Result: The source code of an association class contains appropriate tags for the association class itself, and for each of the association end classes.

To delete an association class:

1. Right-click an association end link, association class, or connector.
2. Choose **Delete Association Class** on the context menu.

Result: The whole association class construct is deleted from the diagram.

Creating an Extension Point

To create an extension point:

1. Right-click the use case element.
2. Choose **Add** ▸ **Extension Point** on the context menu.
3. Type in a name.

Creating an Inner Classifier

This section includes instructions for adding inner classifiers to classes (including Windows classes, such as Windows forms, Inherited forms, User Controls and so on), structures, and modules (collectively, containers) in implementation projects.

You can add inner classifiers to class diagram elements (containers) using the respective context menu for the diagram element in the Diagram or **Model View**s. You can also select a classifier in the **Tool Palette** and click the container element in the **Diagram View** to add the inner classifier to the container element.

Note: Structure elements are available for implementation projects only.

Tip: You can use drag-and-drop or clipboard operations to remove an inner classifier from the container element.

To create an inner classifier by Using the context menu:

1. Right-click the container element.
2. Choose **Add** ▸ **(Inner_classifier_type)**, where **(Inner_classifer_type)** is defined in the table above.

Using cut, copy, and paste:

1. Use the clipboard operations to either cut or copy an existing classifier.
2. Select the container element.
3. Use the clipboard operations to paste the selected classifier into the container element.

Using drag-and-drop:

1. Select an existing classifier in the **Diagram View**.
2. Drag-and-drop it onto a pre-existing container in the Diagram View. A blue border highlights the location that Together recognizes as a valid destination for dropping the inner classifier.

Creating an Internal Structure for a Node

To create an internal structure for a node:

1. Choose the part icon on the diagram **Tool Palette**.
2. Click the valid container (class or collaboration).
3. Repeat these steps to create as many participants as needed.

   Tip: Choose the part icon on the diagram **Tool Palette** while holding down the CTRL key. Each click on a valid container produces a new part.

4. Link the collaborating parts by connectors.
5. Use the **Object Inspector** to set up the properties of the part.

Creating an Internal Transition

To create an internal transition:

1. Select the desired state or activity element on the diagram or in the **Model View**.
2. Right-click the element, and select **Add** ▸ **Internal Transition** on the context menu.

Creating Multiple Elements

You can place several elements of the same type on a diagram without returning to the **Tool Palette** or by using the diagram context menu. Each element will have a default name that can be edited with the in-place editor or in the **Object Inspector**.

To create multiple elements:

1. Holding down the CTRL key, click the **Tool Palette** button for the element you want to create (the button stays down). Release the CTRL key.
2. Click the desired location on the diagram background. The new element is placed on the diagram at the point where you click.
3. Click the next location on the diagram background. The next new element is placed on the diagram.
4. Repeat the previous step until you have the desired number of elements of that type.
5. To stop multiple element creation, click the Pointer **Tool Palette** button or press the ESC key to deselect the element after closing the in-place editor of the last inserted element.

Tip: After making a selection on the **Tool Palette** or doing the first of a multi-draw or multi-placement operation, you can cancel the operation by clicking the Pointer button on the **Tool Palette** or by pressing the ESC key.

Deleting a Diagram

Warning: The project automatically created default diagram for a namespace (package) cannot be deleted.

To delete a diagram:

1. In the **Model View**, select the diagram to be deleted.
2. On the context menu, choose Delete.
3. Confirm deletion, if required.

Result: The diagram is deleted from the project.

Designing a UML 1.5 Activity Diagram

Use the following tips and techniques when you design a UML 1.5 Activity Diagram.

To design a UML 1.5 Activity Diagram, follow this general procedure:

1. Create one or more swimlanes. You can place several swimlanes on a single diagram, or create a separate diagram for each.

   Warning: You cannot create nested swimlanes.

2. Create one or more activities. You can place several activities on a single swimlane, or create a separate swimlane for each.

   Warning: You cannot create nested activities.

3. For convenient browsing, first model the main flow. Next, cover branching, concurrent flows, and object flows.

   Tip: Use separate diagrams as needed and then hyperlink them.

4. Create Start, End, Signal Receipt, and Signal Sending elements for your swimlanes.

   If your activity have several Start points, they can be used simultaneously.

5. Create object nodes. You do not link object nodes to classes on your Class Diagrams. However, you can use hyperlinks for better understanding of your diagrams.
6. Create state nodes for your swimlanes.

   Tip: You can create nested states.

7. Optionally, create a History node.
8. Connect nodes by links.
9. You can optionally create shortcuts to related elements of other diagrams.

Designing a UML 1.5 Component Diagram

Following are tips and techniques that you can use when working with UML 1.5 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.

To design a UML 1.5 Component Diagram, follow this general procedure:

1 Create a hierarchy of Subsystems.

Tip: You can create nested Subsystems.

2 Create a hierarchy of Components. The largest component can be the whole system or its major part (for exam ple, *server application*, *IDE*, *service*).

Tip: You can create nested component nodes. There are two methods for creating a nested component node:

You can select an existing component and add a child component inside.

Alternatively, you can create two separate components and connect them with an Association-Composition link.

3 Create interfaces. Each component can have an interface.

4 Draw links between elements.

5 You can optionally create shortcuts to related elements of other diagrams.

Designing a UML 1.5 Deployment Diagram

Use the following tips and techniques when you design a UML 1.5 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.

To design a UML 1.5 Deployment Diagram, follow this general procedure:

1 Create a hierarchy of Nodes.

Tip: You can create nested Nodes.

2 Create a hierarchy of Components. The largest component can be the whole system or its major part (for exam ple, *server application*, *IDE*, *service*).

Tip: You can create nested Components. There are two methods for creating a nested component:

You can select an existing component and add a child component inside.

Alternatively, you can create two separate components and connect them with an Association-Composition link.

3 Represent how Components resides on Nodes. You can represent this in two ways:

- Use a supports link between the component and node. The supports link is a dependency link with the stereotype field set to support.
- Graphically nest the Component within the Node.

4 Optionally, create Objects.

5 Create Interfaces. Each component can have an interface.

6 Indicate a temporary relationship between a Component and Node. Objects and components can migrate from one component instance to another component instance, and respectively from one node instance to another node instance. In such a case, the object (component) will be on its component (node) only temporarily. To indicate this, use the dependency relationship with a becomes stereotype.

7 You can optionally create shortcuts to related elements of other diagrams.

Designing a UML 1.5 Statechart Diagram

Following are tips and techniques that you can use when working with UML 1.5 Statechart Diagram.

To design a UML 1.5 Statechart Diagram, follow this general procedure:

1. Create Start and End points.
2. Create main states and substates.

   Tip: You can create nested states.

3. Create transitions.
4. Create history nodes.
5. You can optionally create shortcuts to related elements of other diagrams.

To create entry and exit actions:

1. Create an internal transition in the desired state.
2. Double-click the internal transition to enable in-place editing.
3. Rename using the following syntax: `stereotype/actionName(argument)`

   For example: `exit/setState(idle)`

Alternatively, create an internal transition and set the event name, event arguments, and action expression properties using the **Object Inspector** for the internal transition.

Designing a UML 2.0 Activity Diagram

Use the following tips and techniques when you design a UML 2.0 Activity Diagram. Usually you create Activity Diagrams after State Machine Diagrams.

To design a UML 2.0 Activity Diagram, follow this general procedure:

1. Create one or more activities. You can place several activities on a single diagram, or create a separate diagram for each.

   Warning: You cannot create nested activities.

2. Usually activities are linked to states or transitions on State Machine Diagrams. Switch to your State Machine Diagrams and associate the activities you just created with states and transitions.

   Tip: After that you can find that some more activities must be created, or the same activity can be used in several places.

3. Switch back to the Activity Diagram. Think about flows in your activities. You can have an object flow (for transferring data), a control flow, both or even several flows in each activity.
4. Create starting and finishing points for every flow. Each flow can have the following starting points:

   - Initial node
   - Activity parameter (for object flow)
   - Accept event action
   - Accept time event action

   Each flow finishes with a **Activity Final** or **Flow Final** node.

   If your activity has several starting points, they can be used simultaneously.
5. Create object nodes. You do not link object nodes to classes on your Class Diagrams. However, you can use hyperlinks for better understanding of your diagrams.
6. Create action nodes for your flows. Flows can share actions.

   Warning: You cannot create nested actions.

7. For object flows, add pins to actions. Connect actions and pins by flow links.
8. Add pre- and postconditions. You can create plain text or OCL conditions.
9. You can optionally create shortcuts to related elements of other diagrams.

To add an activity parameter to an activity:

1. In the **Tool Palette**, press the **Activity Parameter** button.
2. Click the target activity.

   Or:

   Choose **Add** ▸ **Activity Parameter** on the activity context menu.

Result: An **Activity Parameter** node is added to the activity as a rectangle. Note that the activity parameter node is attached to its activity. You can only move the node along the activity borders.

Note: Activity parameters cannot be connected by control flow links.

Designing a UML 2.0 Component Diagram

Following are tips and techniques that you can use when working with UML 2.0 Component Diagrams. It can be convenient to start creation of a model with Component Diagrams if you are modeling a large system. For example, a distributed, client-server software system, with numerous interconnected modules. You use Component Diagrams for modeling a logical structure of your system, while you use Deployment Diagrams for modeling a physical structure.

To design a UML 2.0 Component Diagram, follow this general procedure:

1. Create a hierarchy of components. The largest component can be the whole system or its major part (for example, *server application*, *IDE*, *service*).

   Tip: You can create nested component nodes. There are two methods for creating a nested component node:

   You can select an existing component and add a child component inside.

   Alternatively, you can create two separate components and connect them with an Association-Composition link.

2. In the hierarchy of components, you can end up by adding concrete classes and instance specifications. You can create them on a Component Diagram directly, or create them on a Class Diagram and put shortcuts on a Component Diagram.
3. Create interfaces. Each component can have a provided interface and a required interface.
4. Optionally, create artifacts. Usually, you describe physical artifacts of your system on Deployment Diagrams. But if some component is closely connected with its physical store, add and link an artifact to a Component Diagram.

   Tip: You can create nested artifacts.

5. Optionally, create ports for your components. You can attach a port to a component and link it with several classes or components inside. In this case, when a message arrives, this port decides which class must handle it.
6. Draw links between elements.
7. You can optionally create shortcuts to related elements of other diagrams.

Designing a UML 2.0 Deployment Diagram

Use the following tips and techniques when you design a UML 2.0 Deployment Diagram. It can be convenient to start creation of a model with Deployment Diagrams if you are modeling a large system that is comprised of multiple modules, especially if these modules reside on different computers. You use Deployment Diagrams for modeling a physical structure of your system, while you use Component Diagrams for modeling a logical structure.

To design a UML 2.0 Deployment Diagram, follow this general procedure:

1. Create a hierarchy of execution environments, devices, and nodes. Execution environments usually represent software environment used to execute your system, such as an operating system. Devices usually represent hardware equipment, such as a printer, a hard disk, or a computer. Nodes represent the rest of physical entities, such as a file.

   Tip: You can create nested execution environments, devices, and nodes. For example, you can add a node inside of an execution environment, or a node inside of a device.

2. Create artifacts.
3. Create deployment and instance specifications. By doing this, you arrange physical locations of objects and other entities of your system.
4. Add operations to artifacts.
5. Once an operation is added, you can define its properties in the **Object Inspector**, which includes parameters, stereotype, multiplicity and more.
6. You can optionally create shortcuts to related elements of other diagrams.

To deploy an artifact to a target node:

1. In the diagram **Tool Palette**, choose the deployment button.
2. Click the artifact to be deployed. The valid source is denoted by a solid frame.
3. Drag-and-drop the deployment link to a target node. The valid target is denoted by a solid frame.

To define parameters of an operation:

1. Select the desired operation in an artifact.
2. In the **Object Inspector**, expand the **General** node and choose **Parameters** field.
3. Click the chooser button to open **Add/Remove Parameters** dialog box.
4. Click **Add**. This creates an entry in the parameters list.
5. Enter the parameter's name, type multiplicity, default value, and direction. Note that parameter type can be selected from the list of pre-defined types, or from the model.
6. Using the **Add** and **Remove** buttons, create the list of parameters.
7. Click **OK** when ready.

Designing a UML 2.0 Sequence or Communication Diagram

Use the following tips and techniques when you design a UML 2.0 Sequence or Communication Diagrams. Usually you create Interaction Diagrams after Class Diagrams.

Whenever an interaction diagram is created, the corresponding interaction is added to the project. Interactions are represented as nodes in the **Model View**.

Note: Presentation of an interaction in the **Model View** depends on the view type defined in the **Model View** options on the default or project group levels. If model-centric mode is selected, an interaction is shown both under its package node and diagram node. If diagram-centric mode is selected, an interaction is shown under the diagram node only.

Note: You can view an interaction in two ways: as a Sequence Diagram, or as a Communication Diagram. So doing, any actions performed with either view are automatically reflected in the other views. Thus, adding or deleting an element in an interaction results in the modification of the corresponding interaction diagram, and vice versa. An interaction diagram contains a reference to the underlying interaction.

Note: Unlike UML 1.5, it is not possible to switch a diagram that already exists from sequence to communication and vice versa. However, it is possible to create a Sequence Diagram and a Communication Diagram based on the same interaction.

To design a UML 2.0 Sequence Diagram, follow this general procedure:

1. Create an interaction use
2. Navigate to a referenced interaction
3. Associate a lifeline with a referenced element
4. Associate a lifeline with a type
5. Define decomposition for a lifeline
6. Repeat the steps to create all required interactions
7. Link the created lifelines by using messages

To create an interaction use:

1. In the diagram **Tool Palette**, choose the **Interaction Use** button.
2. Click on the target lifeline.

   Tip: Alternatively, use the **Add** command on the lifeline context menu in the **Diagram View** or **Model View**.

3. In the **Object Inspector** for the newly created interaction use, choose the Properties tab.
4. In the interaction name field, click the chooser button.

   Tip: Alternatively, just type in the interaction name.

5. In the **Choose Referenced Interaction** dialog box, select the desired interaction from the project or Favorites, and click **OK**.

An interaction use is initially created attached to a lifeline. Further it can be expanded over several lifelines, detached from and reattached to lifelines.

To navigate to a referenced interaction:

1. Right-click on an interaction use that refers to another interaction.
2. On the context menu, choose **Select**.
3. Choose the desired destination on the submenu.

To associate a lifeline with a referenced element:

1. Make sure that your project contains the referenced elements that should be represented by the lifelines.
2. Select the desired lifeline in the **Model View** or the **Diagram View**.
3. In the **Object Inspector**, select the represents field.
4. Click the chooser button.
5. In the **Choose Represented Connectable Element** dialog box, select the desired part from the project or Favorites.
6. Click **OK**.

To associate a lifeline with a type:

1. Select the desired lifeline in the **Model View** or the **Diagram View**.
2. In the **Object Inspector**, select the type field.
3. Click the chooser button.
4. In the **Choose Represented Connectable Element's type** dialog box, select the class that defined the type from the project or Favorites.
5. Click **OK**.

To define decomposition for a lifeline:

1. Select the desired lifeline in the **Model View** or the **Diagram View**.
2. In the **Object Inspector**, select the decomposition field.
3. Click the chooser button.
4. In the **Choose Referenced Interaction** dialog box, select the desired interaction from the project or Favorites.
5. Click **OK**.

> **Tip:** Decomposition, type, stereotype, and referenced element properties are also reflected in the corresponding Communication diagram.

Designing a UML 2.0 State Machine Diagram

Following are tips and techniques that you can use when working with UML 2.0 State Machine Diagram.

To design a UML 2.0 State Machine Diagram, follow this general procedure:

1. Create initial and final nodes.
2. Create main states and substates.
3. Create regions.
4. Create entry and exit points.
5. Create pins.
6. Create transitions.
7. Create history nodes.
8. You can optionally create shortcuts to related elements of other diagrams.

Designing Use Case Hierarchy

Use case diagrams typically represent the context of a system and system requirements.

To design use case hierarchy:

1. Usually, you begin at a high level and specify the main use cases of the system.
2. Next, you determine the main system use cases at a more granular level. As an example, a "Conduct Business" use case can have another level of detail that includes use cases such as "Enter Customers" and "Enter Sales."
3. Once you have achieved the desired level of granularity, it is useful to have a convenient method of expanding or contracting the use cases to grasp the scope and relationships of the system's use case views.

Exporting a Diagram to an Image

To export a diagram to an image:

1. Place the focus on the diagram you want export in the **Diagram View**.
2. Choose **File** ▸ **Export Diagram to Image** on the main menu.
3. Click the drop-down arrow to preview and adjust the zoom settings of the diagram image.
4. Click Save. The file browser dialog box opens.
5. Browse for a location where you wish to save the image.
6. Enter a name. By default, the image file takes on the name given to the diagram in Developer Studio 2006.
7. Select an image format.
8. Click Save.

Grouping Actions into an Activity

Use the following techniques to group actions into an activity:

1. Use the **Tool Palette** buttons
2. Use drag and drop
3. Use the context menu of the activity element

Use the Tool Palette buttons:

1. In the diagram **Tool Palette**, choose to create an activity node.
2. Choose the action button, and click the target activity.

Use drag and drop:

1. Place an action element on the diagram background.
2. Drag and drop the new action on top of an existing activity.

Use the context menu of the activity element:

1. Right-click the target activity.
2. Select **New** ▸ **Action** on the context menu.

Hiding and Showing Model Elements

You can control the visibility of elements on a diagram by using the **Hide** command (available on the context menu for individual diagram elements), and the **Show/Hide** command (available on the diagram context menu).

To hide by using one of the following methods:

1. Open the **Diagram View**.
2. Do one of the following:

- Select the element on the diagram, right-click and choose **Hide** on the context menu.
- Select multiple elements on the diagram using CTRL+Click or by lassoing, and select Hide from the context menu.
- Right-click the diagram background and choose **Hide/Show** on the context menu. The Show Hidden dialog box opens, as discussed below.

To show or hide diagram elements using the Show Hidden dialog box:

1. Right-click the diagram and choose **Show/Hide** on the context menu. The **Show Hidden** dialog box opens.
2. Select the element(s) that you wish to hide from the Diagram Elements list.
3. To add elements in the Diagram Elements list to the Hidden Elements list, do one of the following:

- Double-click the element .
- Click the element once and click Add.
- Select multiple elements using CTRL+Click and click Add.

4. To remove items from the Hidden Elements list do one of the following:

- Double-click the element.
- Click the element once and click **Remove**.
- Select multiple elements using CTRL+Click and click **Remove**.
- To remove all items from the Hidden Elements list, click **Remove All**.

5. Click **OK** to close the dialog box.

Hyperlinking Diagrams

Select Hyperlinks from the diagram context menu to create, view, remove, and browse hyperlinks.

Use the following techniques to create a hyperlink:

1. Create a hyperlink to an existing diagram or element
2. Create a hyperlink to a new diagram
3. Create a hyperlink to an external URL or file
4. Browse hyperlinks
5. Remove a hyperlink

To create a hyperlink to an existing diagram or element:

1. Open an existing diagram or create a new diagram from which to create the hyperlink.
2. Select the element that you want to link to another diagram or element.
3. To link the entire diagram, click the diagram background to deselect all elements.

   Note: Do not select the actual namespace in the **Model View** to create a hyperlink. Rather, expand the namespace node, and select the desired diagram.

4. Right-click and choose **Hyperlinks** ▸ **Edit**. The **Edit Hyperlinks** dialog window (Selection Manager) opens.
5. Select the Model Elements tab to view the pane containing a tree view of the available project contents in the Solution.
6. Select the desired diagram or element from the list, and click **Add**.
7. For element selection, expand diagram nodes in the **Model Elements** tab.
8. To remove an element from the selected list, select the element and click **Remove**.
9. Click **OK** to close the dialog box and create the link.

To create a hyperlink to a new diagram:

1. Open a diagram in the **Diagram View**, or select it in the **Model View**.
2. On the context menu, choose **Hyperlinks** ▸ **To New Diagram**.
3. In the **Add New Diagram** dialog box, select the diagram type, enter the diagram name and click OK.

To create a hyperlink to an external URL or file:

1. Open an existing diagram or create a new diagram from which to create the hyperlink.
2. Select the element that you wish to link to the external document.

   To link the entire diagram, click the diagram background to deselect all elements.

3. Right-click and choose **Hyperlinks** ▸ **Edit**. The **Edit Hyperlinks** dialog box opens.
4. Select the **External Documents** tab to view the Recently used Documents list which contains a list of previously selected files or URLs.
5. To add a file to the Recently used Documents list:

1. Click **Browse**. The **Open file** dialog box opens.

2 Navigate to the desired file and click **Open**.

6 To add a URL to the Recently used Documents list:

1 Click URL.

2 In the dialog box that opens, enter the appropriate URL and click **OK**.

Tip: You can create a hyperlink to an external document by entering a relative URL path.

7 To remove an element from the selected list, select the element and click **Remove**.

8 To clear the Recently used Documents list, click **Clear**.

Note: Items added to the Recently used Documents list are not specific to a single project or project group.

9 Click **OK** to close the dialog box and create the link.

To browse hyperlinks:

1 To view hyperlinks to a diagram, element or external document, right-click on the diagram background or element, and choose Hyperlinks from the context menu. All hyperlinks created appear under the Hyperlinks submenu. On a diagram, all names of diagram elements that are hyperlinked are displayed in blue font. When you select a link from the submenu, the respective element appears selected in the **Diagram View**.

2 Once you have defined hyperlinks for a selected diagram or element, use the context menus to browse to the linked resources.

Note: Browsing to a linked diagram opens it in the **Diagram View** or makes it the current diagram if already open.

Browsing to a linked element causes its parent diagram to open or become current, and the diagram scrolls to the linked element and selects it.

To remove a hyperlink:

1 Open the diagram that displays the link you want to remove.

2 Choose **Hyperlinks** ▸ **Edit** from the diagram or element context menu. The **Edit Hyperlinks** dialog box opens.

3 In the selected list on the right of the dialog, click the hyperlink that you wish to remove.

4 Click **Remove**.

5 Click **OK** to close the dialog box.

Note: To remove a hyperlink from a specific element, select the element first. Then choose **Hyperlinks** ▸ **Edit** on the context menu.

Instantiating a Classifier

In a UML 1.5 design project, you can create an object that instantiates a class or interface from the same or another UML 1.5 design project or any implementation project in the same project group. In an implementation project, you can create an object that instantiates a class or interface from the same project or some UML 1.5 design project or a referenced project. You can create such links by using the **Object Inspector** or by using Dependency links to shortcuts.

To instantiate a classifier:

1. On a UML 1.5 class diagram, choose an object.
2. In the **Object Inspector**, choose the **Instantiates** field.
3. Click the **Chooser** button. The **Choose Type to Instantiate** dialog box opens.
4. In this dialog box, choose a classifier (class or interface).

Tip: Alternatively, draw a Dependency link from this object to a classifier or its shortcut.

Laying Out a Diagram Automatically

To lay out a diagram by using one of the algorithms:

1. Right-click the diagram background.
2. From the context menu, select Layout, and choose a command from the submenu.

   There are several Layout commands on the Layout submenu:

- **Do Full Layout**: Sets the layout of all elements according to the layout algorithm defined for the current diagram.
- **Layout for Printing**:Sets the layout of all elements using the Together algorithm, regardless of the option selected on any level.
- **Route All Links**:Streamlines the links removing bending points.
- **Optimize Sizes**: Enlarges or shrinks all elements on the diagram to the optimal size.

> **Note:** Individual diagram elements also have the Route Links and Optimize Size layout commands on their respective context menus. The Route Links command streamlines the links removing any bending points. The Optimize Size command enlarges or shrinks the element to the optimal size, leaving enough space for its label and any sub elements it may contain.

Tip: To enable layout of the inner substructure in diagrams, check the Recursive option ((level) | Diagram | Layout | General) in the Options dialog window.

To set up the diagram layout:

1. On the main menu choose Tools | Options.
2. On the desired level, select Together | (level) | Diagram | Layout category.
3. Expand the node for the desired algorithm.
4. Specify the algorithm-specific options (if any) and apply changes.

Result: you can observe results of layout tuning when apply one of the Layout commands to the diagram.

The context menu available in the **Diagram View** provides access to the automated layout optimization features in Together.

Linking Another Interaction from an Interaction Diagram

To link another interaction from an interaction diagram:

1. Open an Interaction diagram.
2. Right-click the diagram background and choose **Add** ▸ **Shortcut** on the context menu.
3. Add a shortcut to another interaction in your project.

Moving Model Elements

Create your own layout by selecting and moving single or multiple diagram elements.

You can:

- Select a single element and drag it to a new position.
- Select multiple elements and change their location.
- Manually reroute links.

Note: If you drag an element outside the borders of the **Diagram View**, the diagram automatically scrolls to follow the dragging.

Tip: Manual layouts are saved when you close a diagram or project and restored when you next open it. Manual layouts are not preserved when you run one of the auto-layout commands (Do Full Layout or Optimize Sizes).

To move an element:

1. Select the element or elements to be moved.
2. Drag-and-drop the selection to the target location.

Tip: Right-click and use Cut and Paste. Use the keyboard shortcuts for Cut (CTRL+X), Copy (CTRL+C), and Paste (CTRL+V) operations.

Printing a Diagram

You can print diagrams separately or as a group, or print all diagrams in the project.

To print a diagram:

1 With the diagram in focus in the **Diagram View**, choose **File** ▸ **Print** from the main menu. The **Print diagram** dialog box opens.

2 In the Print Diagrams list box, specify the scope of diagrams to be printed:

- **Active diagram**: To print the currently selected diagram.
- **Active with neighbors**: To print the current diagram and the other diagrams of the same project.
- **All opened**: To print all diagrams currently opened in the Diagram view.
- **All in model**: To print all diagrams within a project group.

3 In the Print zoom field, specify the zoom factor.

4 If necessary, adjust the page and printer settings:

- Click the Print list box and choose Print dialog box to select the target printer.
- Use the Options dialog window (**Together** ▸ **(level)** ▸ **Diagram** ▸ **Print** options) to set up the paper size, orientation, and margins.

Tip: Click Preview to open the preview pane. Use the Preview zoom slider, or Auto Preview zoom check box, as required.

Putting Diagram Files Under Version Control

Together enables you to put your model under the source control in the same way as the other project resources.

The diagram elements in fact belong to the parent default package (namespace) files (`default.txaPackage`). Thus, if editing of a diagram results in adding or deleting elements and changing their properties, you have to check in or check out both diagram file and the corresponding `default.txaPackage` file. If editing only changed the sizes and placement of elements, it is enough to check in or check out the diagram file. To avoid unsynchronized and ambiguous results, each user in a team that works with the same project, should check out and lock the whole project exclusively.

You can use the **History** command to compare changes in diagram files.

Warning: This topic describes the source control actions with regards to modeling. For the detailed information about source control, refer to the relevant Developer Studio 2006 and source control provider documentation.

To put diagram files under version control:

1. Make sure that an integration with a supported source control system is installed on your computer. Source control commands are only enabled when a supported integration is installed.
2. Make sure that your project (project group) is added to the source control.
3. Apply the source control commands to the diagram files by using:

- **StarTeam** menu. This menu contains commands that enable you to add Together diagram files to your source control repository. You can check in, check out, get, exclude, undo check out of, and compare diagram files.
- The **Project Manager** context menu. The context menu commands enable the same actions, and moreover, provide the only way to check in and check out the entire project or project group, and work with the default package or namespace files.

Note: When a project (project group) is added to source control, the **Diagram View** and the **Project Manager** show icons that reflect the status of each model element under source control.

To exclude files from version control:

1. Open the diagram in the **Diagram View**.
2. Choose **StarTeam**. The menu command applies to the diagram that is open in the **Diagram View** and has the current focus.

To check in and check out a project or a project group:

1. Open the **Project Manager**.
2. Right-click the project or project group node.
3. Choose **Check in (Recursive)** or **Check Out** on the context menu.

To check in and check out diagrams:

1. Open the diagram in the **Diagram View**, and choose **StarTeam Check Out** on the main menu. The **Source Control Provider Check out** dialog box opens.

Tip: To retrieve a read-only copy of the latest version of the diagram, choose **StarTeam** ▸ **???** on the main menu.

2 Click **Check out**.

3 After you have finished working with the diagram file, choose **StarTeam** ▸ **Check In** on the main menu.

4 The **Source Control Provider Check in** dialog box opens.

5 Click **Check in**.

Using the Check in command locks the diagram. Diagram elements in the locked diagrams display small lock symbols. You can still edit the locked diagrams, check-out dialog box being automatically invoked on the first attempt to edit or add/remove elements. After that the diagram becomes unlocked.

To undo check out:

1 If you check out a diagram and do not make any changes to it, use the Undo Checkout command. Using this command cancels your check out and removes the writable version of the file from your working folder. The most recent version of your diagram file in the source control repository is copied over your local copy, and if you have made any changes to the local copy since you last checked out the file, they are lost.

2 Open the diagram in the **Diagram View**.

3 Choose **StarTeam** ▸ **Revert**. The menu command applies to the diagram that is open in the **Diagram View** and has the current focus.

To compare diagram versions:

1 Open the diagram in the **Diagram View**.

Warning: To track differences in the versions of a default package file, select the corresponding `default.txaPackage` node in the **Project Manager**.

2 Choose **StarTeam** ▸ **Compare Contents** on the main menu. A dialog box opens.

3 Select a version from the list, and click Diff. The Visual Diff window opens.

Renaming a Diagram

Warning: The project namespace (package) automatically created diagram cannot be renamed.

To rename a diagram:

1. In the **Object Inspector**, double-click the diagram name to initiate the inline editor.
2. Enter a new name.
3. Press Enter.

Alternatively:

1. Select the diagram in the **Model View**.
2. Press F2 or right-click and choose **Rename** on the context menu.
3. Enter a new name.
4. Press Enter.

Result: The diagram is renamed.

Rerouting a Link

To reroute a link:

1. Select a link.
2. Drag and drop the client of supplier end of the link to the desired destination object.
3. To change direction of the link, click a place on the link where you want to reroute the link.
4. Drag the line. Together automatically reshapes the link the way you want.

Tip: Model elements have the **Layout** ▸ **Route All Links** command on diagram context menus.

Resizing Model Elements

Diagram elements can be resized automatically or manually. When new items are added to an element that has never been manually resized, the element automatically grows to enclose the new items.

To resize an element manually:

1. Click an element. The selected element is highlighted with bullets.
2. Drag one of the bullets in the desired direction.

When the element contents change, for example, when members are added or deleted, and the element size is too small to display all members, scroll bars are displayed to the right of compartments.

To optimize a node element size:

1. Right-click an element.
2. Choose **Layout** ▸ **Optimize Size**.

To optimize the elements on an entire diagram:

1. Right-click the diagram background.
2. Choose **Layout** ▸ **Optimize Size**.

Searching Diagrams

Together enables you to use the Find and Replace facilities provided by the Developer Studio 2006 to locate model elements on model diagrams.

To search diagrams:

1. Choose **Search** ▸ **(search command)** to use the find and replace facilities provided by the Developer Studio 2006.
2. You can find the specified string in the specified scope. The function supports case sensitivity, searching for whole words or substrings, using wildcards and regular expressions.
3. Browse the search results.

Searching Source Code for Usages

In addition to the diagram search facility, Together enables you to track how an element or member is used in a source-code project. The **Search for Usages** dialog box enables you to find the references to, and overrides of, the elements and members in implementation projects.

The Search for usages command is available on the context menu of an element in a diagram or in the **Model View**. Note that Search for usages is not available for the design projects.

To search source code for element usages:

1. Right-click an element or a namespace and choose **Search for Usages** on the context menu. The dialog box opens with the selected element specified in the section Element to search.
2. In the Options section, check the following options as required:
   - Usages of elements
   - Usages of members
   - Usages of Declared Classes
   - Implementations
   - Overriding
   - Include usings/imports
   - Skip self
3. Click Search.

The search results are displayed in a tab in the Search for Usages window as a tree view, each node containing all usages of an element in a certain class. Note that each new search adds its own tab to the window.

The Search for Usages window provides a toolbar with the buttons that enable you to expand or collapse the treeview nodes, and repeat the search in the selected tab with the same settings.

The context menu of a search results tab provides the following commands:

Command
Close
Close all
Close all but this

Selecting Model Elements

Most manipulations with diagram elements and links involve dragging the mouse or executing context menu commands on the selected elements.

To select a model element:

1. Open the **Diagram View**.
2. On a diagram:

- Click any element in the diagram to select it.
- To select multiple elements, hold down the CTRL key and click each element individually.
- Click the background and drag a lasso around an area to select all the elements it contains.
- For elements containing members, click on a member to select it.
- To select all elements on a diagram, press CTRL+A. Alternatively, right-click the diagram background and choose **Select All** on the context menu.

Specifying Entry and Exit Actions

You can create entry and exit actions as nodes, or as stereotyped internal transitions.

To specify entry and exit actions using the in-place editor:

1. Create an internal transition in the desired state.
2. Double-click the internal transition to enable in-place editing.
3. Rename the internal transition using the following syntax:

   ```
   stereotype/actionName(argument)
   ```

   For example:

   ```
   exit/setState(idle)
   ```

To specify entry and exit actions using internal transitions:

1. Create the internal transition.
2. Set the event name, event arguments, and action expression properties using the **Object Inspector** for the internal transition.

Using a Class Diagram as a View

Class diagrams can also be used to create subviews of the project.

To use a class diagrams as a view:

1. Create a new class diagram.
2. Create shortcuts to the original diagram to easily and quickly build subset views for easier management.

Tip: Using this feature, you can create views of distributed classes into one diagram, with Together automatically displaying any relationships that the gathered classes may have with each other.

Note: In implementation projects, changes made here also update the source code, keeping diagram and source code in sync.

Using Drag-and-Drop

Drag-and-drop applies to the members as well as to the node elements. You can move or copy members (methods, fields, properties, and so on) by using drag-and-drop in the **Diagram View** or in the **Model View**.

Drag-and-drop functionality from the **Model View** to the **Diagram View** and within the **Model View** works as follows:

- Selecting an element in the **Model View** and using drag-and-drop to place the element onto the diagram creates a shortcut.
- Using drag-and-drop while pressing the SHIFT key moves the element to the selected container.
- Using drag-and-drop while pressing the CTRL key copies the element to the selected container.

Tip: You can also change the origin and destination for links on your diagrams using drag-and-drop.

To move a link to a new destination:

1. Select a link in the **Diagram View**.
2. Hover the cursor over the destination arrow.
3. Drag the arrow and drop it on the new destination. If the destination element is not in view, drag the link in the appropriate direction, and the diagram will scroll with you.

Tip: Follow the same instructions to move the link source to an allowable location.

Using Grid and Other Appearance Options

You can optionally display or hide a design grid on the diagram background and have elements “snap” to the nearest grid coordinate when you place or move them. The grid is configured in the Diagram Appearance options dialog window.

To show grid:

1. Open Options dialog window.
2. Choose the **Together** ▸ **Diagram** ▸ **Appearance** category, Grid group.
3. Adjust the options.

Note: Grid display and snap are enabled by default.

Using the UML in Color Profile

To enable or disable the “UML in color” profile:

1 In the **Options** dialog window, open the **Together** ▸ **(level)** ▸ **Diagram** ▸ **Appearance** category.

Tip: You can enable or disable it on for the project group, project, or diagram level.

2 Set the **Enable UML in color** option to *True* to enable the profile.

3 Optionally, adjust colors used by the profile.

4 Close the **Options** dialog window.

To draw UML nodes in colors:

1 Select or create a classifier.

2 Open the **Object Inspector**.

3 Assign a stereotype that is supported by the “UML in color” profile (for example, *role*).

Result: The classifier changes its color according to the settings in the **Options** dialog window.

Using View Filters

For global control over the diagram view, you can use the filters in the **Options** dialog window.

To enable, disable view filters:

1. Choose **Tools** ▸ **Options** on the main menu.
2. Click the Together folder.
3. Under the **(level)** ▸ **Diagram** node, select View Filters.

Note: The filters shown in the Options dialog window are global filters. To specifically filter classes, you can set the Show members property to False.

To filter classes:

1. In the **Options** dialog window, View Filters page, click the Show members field.
2. Click the drop-down arrow and select False.
3. Click **OK**.

This results in disabling the members, and the inner classifiers (classes, delegates, enumerations, interfaces, and structures).

Since inner classifiers are treated as members of the container element, the following filters do not filter inner classifiers:

View filter
Show classes
Show delegates
Show enumerations
Show interfaces
Show structures

Note: Code-specific elements are available in implementation projects only.

Working with a Collaboration Use

To create a collaboration use:

1. On the diagram **Tool Palette**, choose the Collaboration Use button.
2. Click the target container.
3. Specify the name of the Collaboration Use.

To link to a collaboration type:

1. Select a Collaboration Use element.
2. Specify the type of Collaboration Use using one of the following methods:
   - In the type field of the Collaboration Use in the **Object Inspector**, click the chooser button, and select the collaboration, which the Collaboration Use instantiates, from the Model or Favorites.
   - Next to the name of the Collaboration Use, insert a colon and the name of the collaboration, which the Collaboration Use instantiates.

Result: The type of collaboration use is indicated next to its name.

To unlink from a collaboration type:

1. Right-click the Collaboration Use that has a certain type assigned.
2. On the context menu, choose Unlink Collaboration.

To bind with a role (part):

1. On the diagram **Tool Palette**, choose the Role Binding button.
2. If you hover the mouse over the client collaboration use, the valid client is highlighted with a black ellipse.
3. Drag-and-drop the role binding link to the supplier part. The valid target is highlighted.
4. Type the role name and press Enter to close the in-place editor.

If a collaboration use is associated with a collaboration that contains parts (roles), you can bind them with the parts (roles) of another classifier.

To bind the roles (parts) of the different classifiers via the collaboration use:

1. Create a collaboration use and define its type.
2. Create one or more parts in the collaboration that represents the type.
3. Right-click the target collaboration use and choose Bind new role on its context menu.
4. In the Select Destination dialog box that opens, choose the role to be bound in the target classifier.

Result: A role link is created from the collaboration use to the role in the target classifier. The role link is now marked with the name of the role selected in the collaboration.

Note: Each role can be used for binding only once. With the next invocation of the Bind new role command, the list of available roles no longer displays the ones previously used.

To define an owner:

1. Right-click a collaboration use and choose **Object Inspector** on its context menu.
2. In the owning classifier field of the **Object Inspector**, click the chooser button.
3. In the **Select Owning Classifier** dialog box, navigate to the owner class or collaboration and click OK.

Result: A link is created between the owner as supplier, and the collaboration use as the client. The link is marked with the label `<<represents>>`.

Working with a Combined Fragment

To create a combined fragment:

1. Choose the combined fragment button in the diagram **Tool Palette**, and click on the target lifeline.
2. In the **Type Chooser** dialog box that opens, choose the desired operator from the list of available operators.

Alternatively, you can also create a combined fragment using the context menu of the **Model View**, or **Diagram View**.

To do this, choose the desired lifeline or execution specification in the **Model View**, or in the **Diagram View**. On the context menu of the selection, choose **Add** ▸ **Combined Fragment**. This adds a combined fragment to the target location.

Result: the combined fragment is added to the target lifeline or execution specification. Each new combined fragment has different color, to tell it from the other combined fragments within the same cluster of nested frames.

To create a nested operator:

1. Select the desired combined fragment.
2. In the Operators field of the **Object Inspector**, click the chooser button. Edit **Combined Fragment Operators** dialog box opens.
3. In the Edit operator combobox, select the desired operator. If a certain operator enables parameters, enter the parameter values in the adjacent field. Use commas as delimiters.
4. Click **Add** button. A new line displays below the existing entry in the list of operators, and in the descriptor of the combined fragment.
5. Use **Add** and **Remove** buttons to make up the desired list of the nested operators. Use **Up** and **Down** buttons to specify the proper order of nested operators.
6. Click **Done** to apply changes.

Result: the nested operators are listed in the descriptor of the combined fragment in the specified order.

You can create the nested combined fragments by placing a new combined fragment node inside of an existing one. So doing, each new node is displayed in a different color. The colors are selected at random. You can work with the inner frames same way as with the outer frames: move along a lifeline, spread them over several lifelines, detach and tie frames. Note that drawing a message link from a frame automatically expands it, together with its outer frames, if any.

To create an operand:

1. Select the desired combined fragment in the **Model View** or in the **Diagram View**.
2. On the context menu of the combined fragment, choose **Add** ▸ **Interaction operand**.
3. In the Interaction constraint node select the language to be used for describing constraint. To do this, click the Language drop-down list and choose OCL or plain text.
4. Type the constraint expression.
5. Add as many operands as required.
6. Apply changes.

Result: a new operand is created. Constraint text is displayed in the operand section of the combined fragment.

Working with a Complex State

The techniques in this section pertain to models of particularly complex composite states and substates.

You can resize the main state. You can also create a substate by drawing a state diagram within another state diagram and indicating start, end, and history states as well as transitions.

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

Use the following techniques to create a composite (nested) state:

1. Create a nested substate using drag-and-drop
2. Create a nested substate using the context menu of the state element

To create a nested substate using drag-and-drop:

1. Place a state element on the diagram background.
2. Drag a new state on top of an existing state.
3. Drop a new state.

To create a nested substate using the context menu of the state element:

1. Right-click the state (region) that will be the container.
2. Select **Add** ▸ **State** on the context menu.

Tip: You can nest multiple levels of substates inside one state. For especially complex substate modeling, however, you can find it more convenient to create different diagrams, model each of the substate levels individually, and then hyperlink the diagrams sequentially.

Using the Shortcuts command on the context menu of the diagram, you can reuse existing elements in other state diagrams. Right-click the diagram and choose Add > Shortcuts, navigate within the pane containing the tree view of the available project contents for the project group to the existing diagram, and select its elements, states, histories, forks, and/or joins.

Tip: Using the context menu of the state element, you can also create all of the other subelements that a state can contain.

Tip: Only one History element can be created within one state.

Working with a Constructor

You can create as many constructors in a class as needed.

In design projects, a constructor is created as an operation with the `<<constructor>>` stereotype.

In implementation projects, each new constructor is created with its unique set of parameters. In addition to creating parameters automatically, you can define the custom set of parameters, using the **Object Inspector**.

Tip: You can move, copy and paste constructors and destructors between the container classes same way as the other members.

To define the constructor parameters (implementation projects only):

1. Select the desired constructor in a class.
2. In the **Object Inspector**, click the Params field.
3. In the text field, type the list of parameters in the former type name. Use comma as a delimiter.

Working with a Field

This topic applies to implementation projects only.

In the source code, it is possible to declare several fields in one line. This notation is represented in diagram as a number of separate entries in the Fields section if a class icon. However, you can rename the fields, change modifiers, set initial values and so on, all modifications being applied to the respective field in the diagram icon. Also you can copy and move such fields in diagram (using context menu commands or drag-and-drop), and the pasted field appears in the target container separately.

To rename a field:

1. Choose a field.
2. Enter the new name in the in-place editor of the **Diagram View** or **Model View**, Name text field in the **Object Inspector** or the source code editor.

To define the visibility modifier:

1. Choose a field.
2. Enter the visibility symbol in the in-place editor in the **Diagram View**, or select one from the Visibility combobox in the **Object Inspector**, or edit in the source code editor.

To define the stereotype of a field:

1. Choose a field.
2. Use the in-place editor in the **Diagram View**, or stereotype combobox of the **Object Inspector** or the source code editor.

To define modifiers, initial values, associated objects and so on:

1. Choose a field.
2. Use the **Object Inspector** or the source code editor.

So doing, the model and the source code are kept in sync.

Working with a Provided or Required Interface

To create a provided interface:

1. Create class and interface node elements using the and **Tool Palette** buttons.
2. On the diagram **Tool Palette**, click the provided interface button.
3. Click the client class and drag the mouse to the interface node.

To create a required interface:

1. Create class and interface node elements using the and **Tool Palette** buttons.
2. On the diagram **Tool Palette**, click the required interface button.
3. Click the client class and drag the mouse to the interface node.

Working with a Relationship

You can change the type of an association link.

To draw an association link:

1. Use the association link button on the UML Class Diagram **Tool Palette** to draw association links between diagram elements.
2. The **Object Inspector** enables you to set the link type (association, aggregation, or composition) and the cardinality of the client and supplier.
3. You can also set the link type using the right-click menu of the link. When you create an association link, Together defines a field in the client class (the start of the link).

To set the directed property of an association link:

1. Choose View | **Object Inspector** if the **Object Inspector** is not open.
2. Select a link on the diagram. The properties for the link appear in the **Object Inspector**.
3. In the **Object Inspector**, select the Directed field.
4. Click the drop-down arrow and select the Directed property (*True* or *False*) from the list.

Working with a Tie Frame

To spread a frame to several lifelines:

1. In the diagram **Tool Palette**, choose the Tie Frame button.
2. Click the desired interaction use or combined fragment.
3. Drag-and-drop on the target lifeline.

Result: The frame expands to the target lifeline and is attached to it with a dot.

Working with a UML 1.5 Message

This section describes techniques for working with messages in Sequence and Collaboration diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ.

In a Collaboration diagram, all messages between the two objects are displayed as a generic link line, and a list of messages is created above it. The link line is present as long as there is at least one message between the objects. Messages display in time-ordered sequence from top to bottom of the messages list. In addition to the message links, you can add links that show association and aggregation relationships. These links do not display if you view the diagram as a sequence diagram.

When you draw messages between objects in a sequence diagram, each message is represented by its own link line. Messages in sequence diagrams have more editable properties than messages in collaboration diagrams.

Use the following techniques for messages:

1. Create a self message
2. Reorder a message link
3. Specify creation of an object with a message
4. Specify destruction of an object with a message
5. Specifying a return link by using the Tool Palette (Toolbox)
6. Specify a return link by using the Object Inspector (Properties Window)

To create a self message:

1. Click the **Self Message** button on the **Tool Palette**.
2. For a Sequence diagram, click the lifeline of the object at the point where you want the message to appear. Clicking the object places the message-to-self first on the lifeline.

   For a Collaboration diagram, click the object.

To reorder a message link:

1. Open a diagram.
2. To reorder messages, perform one of the following actions:

   - Drag message links up and down the object lifeline in the **Diagram View**. Reordering automatically updates the message link numbers.
   - Change the **Sequence Number** field in the **Object Inspector**.
   - In the **Diagram View**, use the in-place editor to change the sequence number.

To specify creation of an object with a message:

1. Select a message link in the Sequence diagram.
2. In the **Object Inspector** of the message link, click the **Creation** field.
3. Choose *True* from the list box.

Result: The message link points to the recipient object icon rather than to its lifeline. The created object moves downward along the lifeline to show that it exists at a point later in time from its creator.

By default, the Creation property is set to False in the Properties Window.

To specify destruction of an object with a message:

1. Select a message link in the Sequence diagram.
2. In the **Object Inspector** of the message link, click the Destruction field.
3. Choose True from the list box.

Result: The object is destroyed.

By default, the **Destruction** property is set to *False* in the **Object Inspector**.

To specifying a return link by using the Tool Palette (Toolbox):

1. Click the **Return link** button in the **Tool Palette**.
2. On the sequence diagram, click the object lifeline element at the supplier end of the message link to draw the return link.

To specify a return link by using the Object Inspector (Properties Window):

1. Select the message link on the sequence diagram.
2. Choose View | **Object Inspector** on the main menu or press F4.
3. In the **Object Inspector**, click the drop-down arrow for the **Return Arrow** field and select *True*.

Working with a UML 2.0 Message

This section describes techniques for working with messages in sequence and communication diagrams. Although the two diagram types are equivalent, the techniques for dealing with messages differ.

Use the following technique for UML 2.0 messages:

1. Show or hide reply message
2. Nest messages
3. Create a message from a lifeline back to itself
4. Create a message link that corresponds to an operation call

To show or hide reply message:

1. Select a call message in an interaction diagram.
2. In the Link tab of the **Object Inspector**, check or clear show reply message.

To nest messages:

1. You can nest messages by originating message links from an execution specification. The nested message inherits the numbering of the parent message. For example, if the parent message has the number 1, its first nested message is 1.1.
2. It is also possible to create message links back to the parent execution specifications.

To create a message from a lifeline back to itself:

1. Click the **Message** button on the **Tool Palette**.
2. In a Sequence diagram, click twice the lifeline in the place where you want this message to appear.

   In a Communication diagram, click twice the lifeline anywhere.

To create a message link that corresponds to an operation call:

1. Create an interaction.
2. Create a message link between two lifelines in the interaction.
3. Open the **Link** tab of the message link **Object Inspector**.
4. In the signature field, click the browse button.
5. In the Model or Favorites, select the desired operation.
6. Click **OK**.

The message link is named according to the name of the operation.

Working with an Instance Specification

You can instantiate a classifier using the **Object Inspector** or the in-place editor.

Use the following techniques with an instance specification:

1. Instantiate a classifier using the **Object Inspector**
2. Instantiate a classifier using the in-place editor
3. Define the features of an instance specification
4. Add a slot to an instance specification element
5. Associate a slot with a structural feature
6. Set the slot value
7. Define the slot stereotype

To instantiate a classifier using the Object Inspector:

1. Select an instance specification in your diagram.
2. In the General node of the **Object Inspector**, select the instantiates field.
3. Click the chooser button.
4. In the **Choose Class or Interface for Type** dialog box, select the classifiers from the available contents, using the **Add** and **Remove** buttons.
5. Click **OK** when ready.

To instantiate a classifier using the in-place editor:

1. Select an instance specification in your diagram.
2. Press F2 to open the in-place editor. Alternatively, click twice on the instance specification name.
3. Type the name of an existing classifier, delimited by a colon, next to the instance specification name. For exam ple, `InstanceSpecifcation1:Class1`.
4. Press Enter.

To define the features of an instance specification:

1. Insert slots into an instance specification element.
2. Associate the slots with the attributes of the instantiated classifiers.
3. Set value, and define the slot stereotype.

To add a slot to an instance specification element:

1. Add an instance specification element to your diagram.
2. Right-click the instance specification element on your diagram and choose **New** ▸ **Slot** on the context menu.

To associate a slot with a structural feature:

1. Select a slot in an instance specification element.
2. Expand the **General** node of the **Object Inspector**.
3. In the defining feature field, click the chooser button.
4. In the **Choose Attribute for Defining Feature** dialog box, select the desired attribute and click **OK**.

To set the slot value:

1. Choose a slot.
2. Do one of the following:

- In the **Object Inspector** of the slot, type the desired string in the value field.
- Invoke the in-place editor for the slot and type the value next to the slot name, delimited by a equal sign.

To define the slot stereotype:

1. In the **Object Inspector** of the slot, expand the **General** node.
2. In the **Stereotype** filed, enter the stereotype value.

Working with an Interface

To create an interface:

1. Create a class and an interface node elements using the and **Tool Palette** buttons.
2. On the diagram **Tool Palette**, click the generalization link button.
3. Click the client class and drag the mouse cursor to the interface node.

To hide an interface:

1. Select an interface.
2. Right-click and choose **Object Inspector** on its context menu.
3. Expand the **View** node.
4. Check the invisible option.

You can hide all interfaces by disabling the **Show Interfaces** view filter.

Working with an Object Flow or a Control Flow

You can create control flow or object flow as an ordinary link between the two node elements. The valid nodes are highlighted when the link is established.

You can scroll to the target element if it is out of direct reach, or you can use the context menu command to avoid scrolling.

There are certain limitations stipulated by UML 2.0 specifications:

- Object flow link must have an object at least on one of its ends.
- It is impossible to connect two actions with an object flow except through an output pin on the source action.
- Control flow link may not connect objects and/or activity parameters.

Use the following techniques with an object flow or a control flow:

1. Create a flow
2. Create a fork or a join
3. Create a decision or a merge

To create a flow:

1. Right-click the source element of the flow.
2. On the context menu, choose **Add** ▸ **Control Flow** or **Add** ▸ **Object Flow**. The **Choose Destination** dialog box opens.
3. In the **Choose Destination** dialog box, select the target and click OK. Note that the OK button is only enabled when the valid target is selected.

To create a fork or a join:

1. Identify the actions involved. If necessary, place all of the actions on the diagram first. Lay them out as desired.
2. Place either a fork or a join on the diagram. Resize as needed.
3. If depicting multiple sources, draw control flow from each of the source actions to the join, and from the join to the target action. If depicting multiple targets, draw control flow from the source action to the fork, and from the fork to each of the target actions.

To create a decision or a merge:

1. Identify the actions involved. If necessary, place all of the actions on the diagram first. Lay them out as desired.
2. Place either a decision or a merge on the diagram. Resize as needed.
3. If merging multiple actions, draw control flow from each of the source actions to the merge, and from the merge to the target action. If making a decision, draw control flow from the source action to the decision, and from the decision to each of the target actions.

Working with User Properties

User properties are created by means of the **User Properties** command. The **User Properties** command is available on the context menus of the diagrams and diagram elements both in the **Diagram View** and the **Model View**. Once created, the user properties can be viewed and edited in the **Object Inspector** under the User Properties category.

To create user properties:

1. In the **Diagram View** or the **Model View**, select the desired diagram or model element.
2. On the context menu, choose **User Properties**.
3. In the **Add/Remove user properties** dialog box, click the Add button. A new entry, consisting of the Name and Value fields, is added to the properties list.
4. In the new entry, enter the property name and value.
5. Using the Add and Remove buttons, make up the list of user properties.
6. Click **OK** when ready.

Result: The User Properties category appears in the **Object Inspector**.

Zooming a Diagram

Use the diagram context menu to obtain the required magnification in the **Diagram View**.

To specify the magnification in the Diagram View:

1. Right-click the diagram background.
2. Select **Zoom** on the context menu.
3. Choose a command from the submenu.

Together Documentation Generation Procedures

Configuring the Documentation Generation Facility

To define the documentation title, header, footer and other specific settings, use the **Options** dialog window.

Descriptions of the options are provided in the **Options** dialog window. You can also find their descriptions in this online help.

To configure the documentation generation facility:

1. On the main menu, choose **Tools** ▸ **Options** ▸ **Together** ▸ **(level)** ▸ **Generate Documentation**.
2. Under the General category, enter the documentation title, window title, header, and footer.
3. Set the User Internal Browser option to choose to open the generated documentation in an external browser or in the Developer Studio 2006 internal browser. By default, documentation opens in your external browser.
4. Under the Include category, select the visibility modifiers for classes and members to be included in the generated documentation.
5. Under the Navigation category, set up the options for generating navigation bar, index, class hierarchy, and help link.

Generating Project Documentation

To generate project documentation:

1. Select project name, namespace or diagram in the **Model View**.
2. Select **Tools** ▸ **Generate Documentation** on the main menu. Alternatively, right-click the selection and choose **Generate Documentation** on the context menu.
3. In the **Generate Documentation** dialog box that opens, select your preferred Scope and Options settings.
4. Click **OK** to generate documentation. By default, the Generate Documentation wizard creates documentation for your entire project.

Together Object Constraint Language (OCL) Procedures

Creating a Guard Condition for a Transition

To create a guard condition for a transition:

1. Select a transition in the diagram.
2. Under the General node of the **Object Inspector**, click the **Guard** field.
3. Type the condition expression and apply changes.

Creating a State

To create a state:

1 Using the **Tool Palette** buttons: On the diagram **Tool Palette**, choose to create a state node. Click an appropriate place on your diagram.

Alternatively:

Using the context menu of the diagram: Right-click the diagram background. Select **Add** ▸ **State** on the context menu.

> **Note:** You can place a state inside of the existing state. It is possible to hide individual states. For example, you might want to hide the content of composite states for better understanding of the whole diagram.

2 When a new state is placed on a diagram, you can use the **Object Inspector** to adjust its properties, including:

- Configure standard properties of the element.
- In the State Invariant field, select the language of the expression from the Language list box. The possible options are OCL and plain text.
- In the Properties page, configure the behavior of the state by setting or viewing the following additional properties:

Field	Description
Composite	Set to True if there is one or more regions in this state (not editable)
Orthogonal	Set to True if there are two or more regions in this state (not editable)
Simple	Set to True if there are no regions in this state (not editable)
Do activity	Specify the activity to be performed during execution of the current state by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project
Entry	Specify the activity to be performed when the current state starts executing by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project
Exit	Specify the activity to be performed when the current state finishes executing by using the **Object Inspector**. This activity may be selected from any Activity diagram of the project

In the edit field below the list box enter the OCL expression for this state.

Creating a State Invariant

To create a state invariant as an OCL comment:

1. On the diagram **Tool Palette**, choose the state invariant button.
2. Click the target lifeline or execution specification.

   Tip: Alternatively, use the **Add** ▸ **State invariant** command on the context menu of a lifeline or an execution specification.

3. In the **Object Inspector** of the state invariant, select the General node.
4. In the Invariant kind field, choose OCL expression from the drop-down list. The shape of the state invariant diagram element changes to braces.
5. In the OCL invariant node that adds to the Property Browser, select the language of the comment from the Language drop-down list. The possible options are OCL and plain text.
6. Type the text and apply changes.

To connect a state invariant to a state:

1. On the diagram **Tool Palette**, choose the state invariant button.
2. Click the target lifeline or execution specification.
3. In the **Object Inspector** of the state invariant, select the General node.
4. In the Invariant kind field, choose States/Regions from the drop-down list.
5. In the States/Regions field, click the chooser button.
6. In the **Choose States and/or Regions** dialog box, select the desired states and/or regions from the model, using the Add button.
7. Click OK when ready.

   Tip: Alternatively, type the state or region name. If the state or region belongs to a different package, specify its fully-qualified name.

Creating an OCL Constraint

To create an object constraint and link it with the context:

1. In the Class/package diagram **Tool Palette**, choose the Constraint button and click the diagram background. The note element appears with the OCL editor activated.
2. Type the constraint expression.
3. Close the OCL editor.
4. In the diagram **Tool Palette**, choose the button, and link the constraint node with the respective design element.

Tip: The constrained attribute should actually exist in the context. Otherwise the constraint will be marked as invalid.

Alternatively, follow these steps:

1. In the **Model View** or in the diagram, right-click an element for which a constraint should be created.
2. Choose Constraints.
3. In the Add / Remove constraints dialog box, click Add.
4. Enter the constraint:
   - In the Name field, enter the constraint name.
   - In the Language field, choose OCL or text from the list box.
   - In the Constraint field, enter the constraint text.
5. Add as many constrains as needed.
6. Click **OK** when ready.

Editing an OCL Expression

To activate the OCL Editor:

1. Double-click a constraint element or OCL property, or select a constraint element and press F2. The OCL Editor window opens.
2. Edit an expression.
3. Use the green button to apply changes and close the OCL Editor. Use the red button to discard changes and close the OCL Editor.

Showing and Hiding an OCL Constraint

To hide an individual constraint:

1. Right-click a constraint in the diagram.
2. Choose Hide.

To hide multiple constraints:

1. Right-click the diagram background.
2. Choose Show/Hide.
3. In the Show Hidden dialog box, select the desired constraints in the Diagram Elements list.
4. Click Add.

To reveal the hidden constraints:

1. Right-click the diagram background.
2. Choose Show/Hide.
3. In the Show Hidden dialog box, select the desired constraints in the Hidden list.
4. Click Remove.

Working with a Combined Fragment

To create a combined fragment:

1. Choose the combined fragment button in the diagram **Tool Palette**, and click on the target lifeline.
2. In the **Type Chooser** dialog box that opens, choose the desired operator from the list of available operators.

Alternatively, you can also create a combined fragment using the context menu of the **Model View**, or **Diagram View**.

To do this, choose the desired lifeline or execution specification in the **Model View**, or in the **Diagram View**. On the context menu of the selection, choose **Add** ▸ **Combined Fragment**. This adds a combined fragment to the target location.

Result: the combined fragment is added to the target lifeline or execution specification. Each new combined fragment has different color, to tell it from the other combined fragments within the same cluster of nested frames.

To create a nested operator:

1. Select the desired combined fragment.
2. In the Operators field of the **Object Inspector**, click the chooser button. Edit **Combined Fragment Operators** dialog box opens.
3. In the Edit operator combobox, select the desired operator. If a certain operator enables parameters, enter the parameter values in the adjacent field. Use commas as delimiters.
4. Click **Add** button. A new line displays below the existing entry in the list of operators, and in the descriptor of the combined fragment.
5. Use **Add** and **Remove** buttons to make up the desired list of the nested operators. Use **Up** and **Down** buttons to specify the proper order of nested operators.
6. Click **Done** to apply changes.

Result: the nested operators are listed in the descriptor of the combined fragment in the specified order.

You can create the nested combined fragments by placing a new combined fragment node inside of an existing one. So doing, each new node is displayed in a different color. The colors are selected at random. You can work with the inner frames same way as with the outer frames: move along a lifeline, spread them over several lifelines, detach and tie frames. Note that drawing a message link from a frame automatically expands it, together with its outer frames, if any.

To create an operand:

1. Select the desired combined fragment in the **Model View** or in the **Diagram View**.
2. On the context menu of the combined fragment, choose **Add** ▸ **Interaction operand**.
3. In the Interaction constraint node select the language to be used for describing constraint. To do this, click the Language drop-down list and choose OCL or plain text.
4. Type the constraint expression.
5. Add as many operands as required.
6. Apply changes.

Result: a new operand is created. Constraint text is displayed in the operand section of the combined fragment.

Together Pattern Procedures

Adding Participants to the Patterns as First Class Citizens

Patterns as First Class Citizens are represented by the GoF patterns. When such patterns are applied, the elements are created with the standard number of participants. However, you can add allowed participants to the existing pattern object. If you add participants, links between the pattern object and the new participants are created.

To add a participant to a GoF pattern:

1. Select the oval pattern element in the **Diagram View** or **Model View**
2. Right-click on the pattern element choose **Add** from the context menu. The submenu presents the list of allowed participants.
3. Choose the required participant from the submenu.
4. In the **Pattern Action Wizard**, specify the name of the new participant, and click **OK**.

Tip: If the participant with the specified name already exists, it is reused.

Assigning Patterns to Shortcuts

You can associate a pattern with one or more shortcuts, located in the various virtual folders.

To assign a pattern to a shortcut:

1. In the **Virtual pattern trees** section of the **Pattern Organizer**, select the desired shortcut.
2. Right-click and choose **Assign Pattern**. The **Pattern Registry** opens.
3. In the **Pattern Registry**, select the pattern to be assigned to the selected shortcut, and click **OK**.
4. In the **Properties** section of the**Pattern Organizer**, edit the shortcut name and visibility as required
5. Save the changes.

Copying and Pasting Shortcuts, Folders or Pattern Trees

To copy and paste a folder or a shortcut:

1. In the **Virtual pattern trees** section, select a shortcut, folder or pattern tree to be copied.
2. Right-click the node and choose **Copy** on the context menu.

   Tip: Alternatively, press CTRL+C

3. Right-click the destination node and choose **Paste** on the context menu.
   Alternatively, press CTRL+V
4. Save changes.

Creating a Folder

Use virtual folders to logically organize patterns in the pattern trees.

To create a new folder:

1. In the **Pattern Organizer**, select the target node in the **Virtual pattern trees** section.
2. Right-click this node and choose **New Folder**. The *New Folder* node is added.
3. In the **Properties** section, edit the **Name** and **Visible** fields as required.

Creating a Link by Pattern

Together makes it easy for you to apply patterns when creating links. To create links during modeling, you can use the **Link by Pattern** button in the diagram **Tool Palette**. The **Link by Pattern** button launches the **Pattern Wizard** dialog displaying the available patterns.

To create a link by pattern:

1. Click the **Link by Pattern** button in the diagram **Tool Palette**. The button stays down.
2. Click the source element on the diagram.
3. Drag to the destination element and drop when the second element is highlighted. The **Pattern Wizard** opens.
4. In the **Pattern Wizard** window, select the pattern that you want to apply for the new link, define its properties and click **Finish**.

Creating a Model Element by Pattern

You can apply patterns explicitly using the **Node by Pattern** button in the **Tool Palette** or by using the right-click menu command Create by Pattern. Whenever you create an element on a diagram using one of the toolbar buttons, you are applying a default pattern that is connected to the selected button.

To create model elements by pattern:

1. On the diagram **Tool Palette**, choose the **Node by Pattern** button.
2. Click the container, where you want to add an element by pattern. This can be either the diagram background or a node element. **Pattern Wizard** opens.

   Tip: Alternatively, right-click the target container and choose **Create by Pattern** on the context menu.

3. In the **Pattern Wizard** select the desired pattern, modify its properties and click **OK**.

Creating a Pattern

You can use existing diagram elements as the basis to create custom patterns. The newly created patterns are stored in the **Pattern Registry**. They become visible in the pattern tree of the **Pattern Organizer** and can be used to generate design elements in diagrams.

To create a pattern:

1. Select one or more elements on a diagram.
2. Right-click and choose **Save as Pattern** on the context menu of the selection. The **Create Pattern Wizard** opens.
3. On the **General** page of the wizard enter the following information:
   - In the **File** field specify the target XML file name.
   - In the **Name** field specify the name of the new pattern.
   - Optionally, enter the pattern description in the **Description** field
   - Optionally, check **Create Pattern Object** check box. Selecting this option allows you to use your pattern as a First Class Citizen. This means that an oval pattern element will display on your diagrams when applying the pattern.
   - Click **Next**.
4. On the **Pattern Parameters** page of the wizard:
   - Use the in-line editor to modify the parameters as required.
   - Set the **Use Existent** property for the pattern. If this value is checked, existing elements on the diagram are reused when you apply the pattern. This means that whenever you apply a pattern, a new element is not created if there is an element with the same name and metatype in the target container . If you clear the**Use Existent** property, then new elements are created.
   - Click **Next**.
5. In the **Select tree folder** page that displays the current patterns structure, choose the target folder, and click **OK**.

Result: The new pattern is added to the specified folder. This pattern is visible in the pattern tree and can be used to generate design elements.

Creating a Shortcut to a Pattern

In the Pattern Organizer you are working with shortcuts, not with the actual patterns. Because of this, shortcuts to the same pattern may be included in several folders.

To create a new shortcut to a pattern:

1. In the **Pattern Organizer**, select the topmost target node.
2. Right-click this node and choose **New Shortcut**. The **Pattern Registry** opens.
3. In the **Pattern Registry**, select the pattern to be assigned to the new shortcut, and click **OK**
4. When the **Pattern Organizer** prompts you to save changes in the **Pattern Registry**, click **Yes**.

Creating a Virtual Pattern Tree

The **Pattern Organizer** enables you to logically organize patterns using virtual trees, folders and shortcuts. Under a tree node you can create virtual folders and shortcuts to patterns.

To create a new pattern tree:

1. In the **Pattern Organizer**, select the topmost **Patterns** node.
2. Right-click this node and choose **New Pattern Tree**. The *New Pattern Tree* node is added.
3. In the **Properties** section, edit the **Name** and **Visible** fields as required.

Deleting Patterns as First Class Citizens from the Model

You can delete elements of the patterns as First Class Citizens (GoF patterns), using both the **Diagram View** and the **Model View**. If you delete elements, they are removed from the diagram and from the model.

To delete a GoF pattern with participants:

1. In the **Diagram View** or **Model View**, select the oval pattern element to be deleted.
2. On the context menu of the selection, choose the **Delete with Participants** command.
3. Confirm deletion.

Deleting shortcuts, folders or pattern trees

To delete a node from the Pattern Organizer:

1. In the **Virtual pattern trees** section, select a shortcut, folder or pattern tree to be deleted.
2. Right-click the node and choose **Delete** on the context menu.
   Alternatively, press DELETE key
3. Save changes.

Editing Properties

Properties of the virtual trees, folders and shortcuts are displayed in the properties section of the **Pattern Organizer**. Using the toolbar buttons, you can choose the properties presentation: in expandable nodes, or in alphabetical order. The **Name** and **Visible** properties are editable. Changes are applied when the edited field looses the focus, or the Enter key is pressed. The node name in the tree view changes accordingly.

To edit properties of a tree, shortcut or folder:

1. Select a node in the **Virtual pattern trees** section.
2. In the **Properties** section, edit the **Name** property, using the text field.
3. In the **Properties** section, edit the **Visible** property, using the drop-down list.

   Tip: The **Visible** property applies to shortcuts only. If **Visible** is set to Visible, the shortcut is displayed in the Pattern Wizard. Otherwise, it is not visible. If a folder does not contain any visible shortcuts, it is also hidden in the Pattern Wizard.

4. Save changes.

Exporting a Pattern

You can create patterns and export them to the specified location.

To export a pattern:

1. In the **Pattern Organizer** window, expand the pattern tree and locate the folder to be exported.
2. Right-click the selected folder and choose **Export folder**.
3. In the **Select path to export** dialog box, navigate to the desired location, and click **Save**.

Importing a Legacy Pattern

You can reuse patterns created in the different versions of Together. Upon starting Together, the available storage is scanned for patterns, and all the encountered patterns are included in the Pattern Registry. However, they are not available for usage unless you manually create shortcuts to these patterns in the **Pattern Organizer**.

To reuse a custom pattern, follow this general procedure:

1. Copy your legacy patterns to the folder that stores patterns in your product installation folder.
2. After the product startup, Pattern Registry automatically registers all available patterns.
3. Open the **Pattern Organizer**,
4. In the **Pattern Organizer** window:
   - Locate the target folder for the patterns in question, or create a new folder.
   - Create a new shortcut.
   - Assign the desired pattern to this shortcut.
5. Save changes.

Opening the Pattern Organizer

The **Pattern Organizer** enables you to logically organize patterns (using virtual trees, folders and shortcuts), and view and edit the pattern properties.

To open the Pattern Organizer:

1. On the main menu, choose **Tools** ▸ **Pattern Organizer**.
2. Result: The **Pattern Organizer** window opens.

Saving Changes in the Pattern Registry

If you have changed the contents of the **Pattern Registry** using the **Pattern Organizer** (created new shortcuts, exported or created shared folders), these changes are synchronized with the Registry automatically. When you close the **Pattern Organizer**, you are prompted to save changes. Each time you start Together, the contents of the available storage is scanned for patterns. The contents of the registry is synchronized with the actual availability of the pattern folders. If you have made changes to the patterns outside the Organizer, these changes will be synchronized when Together is started.

To save changes in the Pattern Registry:

1. In the **Pattern Organizer** click **Close** button. The dialog window opens prompting you to save changes in the pattern registry.
2. Click **Yes** to confirm.

Tip: Alternatively, open the **Pattern Registry** dialog, and click **Synchronize**.

Sharing Patterns

You can store patterns in the shared locations, to facilitate team development. The **Pattern Organizer** enables access to the shared patterns if the paths to these patterns are included in the list of Shared Pattern Roots. being included in the list, patterns from the shared location become visible in the Custom Patterns node of the patterns tree.

To create shared patterns:

1. Export the desired patterns to a shared location.
2. In the **Pattern Organizer**, click **Edit Shared Patterns Roots**. **Shared Patterns Roots** dialog opens.
3. In the List of Shared Patterns Roots, click **Add**. **Select Shared Pattern Tree** dialog opens.
4. In the **Select Shared Pattern Tree** dialog locate the folder that contains the desired patterns, select the `Shortcut Registry.xml` file and click **Open**. The path is added to the list of shared pattern roots.
5. Edit the list using **Add** and **Remove** buttons.
6. Click **OK** when ready.

Sorting Patterns

While working with the **Pattern Organizer**, the logical trees, folders, and shortcuts may be displayed in an arbitrary order. You can sort nodes alphabetically within the container node, using the **Sort Folder** command.

To sort patterns the Pattern Organizer:

1. In the **Virtual pattern trees** section, select the node to be sorted.
2. Right-click the node and choose **Sort Folder** on the context menu.
3. Save changes.

Using the Pattern Organizer

The **Pattern Organizer** enables you to:

- Create logical pattern trees and folders
- Create shortcuts to patterns
- Assign patterns to shortcuts
- Copy, paste and delete trees, folders and shortcuts
- Save changes in the Pattern Registry

Using the Pattern Registry

The **Pattern Registry** is only available from the Pattern Organizer context menu, when you create a new shortcut, or assign a pattern to a shortcut. In the Pattern Registry you can filter patterns by category, metaclass, diagram type, language or status of registration.

To open the **Pattern Registry**, do one of the following:

- Right-click a folder and choose **New shortcut**.
- Right-click a pattern shortcut and choose **Assign Pattern**.

To filter patterns in the Pattern Registry:

1. In the Filters section of the Pattern Registry dialog window, click the attribute to filter the patterns.
2. Select the desired value from the drop-down list.

Using the Stub Implementation Pattern

To create an inheritance link with stub implementation using the Link by Pattern button:

1. Click the Link by Pattern button in the **Tool Palette**.
2. Click the source class and drag-and-drop the link to the destination class or interface. The Pattern Wizard dialog opens.
3. In the Pattern Wizard, expand the Standard folder and select Implementation link and stub.
4. Click OK to complete the stub implementation. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

To create an inheritance link with stub implementation using the Node by Pattern button:

1. Click the Node by Pattern button in the **Tool Palette**.
2. Select the source class on the diagram. The Pattern Wizard opens.
3. In the Pattern Wizard, expand the Standard folder, and select Implementation link and stub.
4. In the Pattern Properties pane on the right of the Pattern Wizard, click the information button to the right of the Supplier field. The Select Supplier dialog opens.
5. Select the destination class or interface from the treeview of available contents and click Ok.
6. Click OK to complete the stub implementation and close the Pattern Wizard. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

To create an inheritance link with stub implementation using the Create by Pattern context menu:

1. Right-click the source class on the diagram and choose Create by Pattern from the context menu. The Pattern Wizard opens.
2. In the Pattern Wizard, expand the Standard folder and select Implementation link and stub.
3. In the Pattern Properties pane on the right of the Pattern Wizard, click the information button to the right of the Supplier field. The Select Supplier dialog opens.
4. Select the destination class or interface from the treeview of available contents and click Ok.
5. Click OK to complete the stub implementation and close the Pattern Wizard. The inheritance link is created and the stubs for the inherited methods are generated in the source class.

Note: You can find the Stub implementation pattern on the context menu of classes that inherit from an interface or an abstract class. This pattern is also available in the Pattern Wizard by clicking the Node by Pattern button in the **Tool Palette**, or by using the Create by Pattern context menu for a class. Use the Stub implementation pattern if you already have an inheritance/generalization link drawn on the diagram and you want to copy the methods to the source class.

To create a stub implementation using the class context menu:

1. Right-click a class that inherits from an interface or an abstract class.
2. Choose Stub Implementation from the context menu.

To create a stub implementation using the Node by Pattern button:

1. Click the Node by Pattern button in the **Tool Palette**.
2. Select the source class on the diagram. The Pattern Wizard opens.
3. In the Pattern Wizard, expand the Standard folder, and select Stub implementation.
4. Click OK to complete the stub implementation and close the Pattern Wizard. The stubs for the inherited methods are generated in the source class.

To create a stub implementation using the Create by Pattern context menu:

1. Right-click the source class on the diagram and choose Create by Pattern from the context menu. The Pattern Wizard opens.
2. In the Pattern Wizard, expand the Standard folder, and select Stub implementation.
3. Click OK to complete the stub implementation and close the Pattern Wizard. The stubs for the inherited methods are generated in the source class.

Together Project Procedures

Activating Together Support for Projects

This topic describes how to activate Together support for a project individually.

Tip: You can also force Together to activate support automatically for all new or currently open projects by adjusting General options.

To activate Together support follow these steps:

1. Switch to a desired project or project group.
2. Choose **Project** ▸ **Together Support** on the main menu.

   Tip: Alternatively, choose **Together Support** from the project node context menu in the **Project Manager** or **Structure View**.

   Result: The **Model Support** dialog box opens showing the list of projects within the current project group.
3. In the **Model Support** dialog box, check the flags for those projects where you need modeling.
4. Click **OK**.

Result: The **Model View** displays the models for each of the selected projects. In the **Project Manager**, `ModelSupport_%PROJECTNAME% ModelSupport` folder is added to each of the selected projects.

To deactivate Together support, follow the above procedure, but uncheck the flags for those projects of a project group that do not need modeling.

Creating a Project

To create a Together project:

1. On the main menu, choose **File** ▸ **New** ▸ **Other**. The **New Project** dialog box opens.
2. From the **Project Types** pane, choose the desired project category.
3. From the Templates pane, choose the desired project template.
4. Enter the project name, location and other parameters as required by the **New Project** dialog box.
5. Click **OK**.
6. Follow the procedure provided by the **New Project Wizard**.
7. In the **Project from MDL** wizard, click the Add Folder button and choose the desired source folder from the file system. Use the Remove and Remove all buttons to make up the list of model files.

Result: A project of the selected type is created in the specified location.

For design project, `.bdsproj` `.tgproj` file is created in the specified project root. The default package and diagram are created.

For implementation project, if Together support is enabled, `.bdsproj` file is created in the specified project root, the default namespace and diagram are created.

Exporting a Project to XMI Format

To export a project to XMI format:

1. In the **Model View**, right-click the root project node, and choose Export Project to XMI, or choose **File** ▸ **Export Project to XMI** on the main menu. The **XMI Export** dialog box opens.
2. In the Select XMI Type groupbox, select the xml/uml version you wish the file to support. You can select from the available XMI Type choices:
   - XMI for UML 1.3 (Unisys Extension)
   - XMI for UML 1.3 (Unisys Extension, Recommended for TCC), default value
   - XMI for UML 1.3 (Unisys Extension, Recommended for IBM Rational Rose)
3. Click the drop-down arrow to select an appropriate XMI encoding requirement. The default value is UTF-8.
4. Specify the export destination. You can include the path as well as the name of the file (.xml) which will be created, or you can accept the default: `(project folder)\out\xmi\(project name).xml`
5. Click Export. If the destination directory does not exist, a confirmation dialog asks if you want to create it.
6. Click Yes.

Result: The created XML file is added to the specified location.

Importing a Project Created in TCC or TAR

You can use the following steps to migrate your existing Together ControlCenter diagrams and source code to Developer Studio 2006.

To make it possible to use the Together ControlCenter models in Developer Studio 2006, Together ControlCenter supports the packaging namespace organization for C# projects. This means that you can optionally require automatic settings of namespaces for C# classifiers in the Together ControlCenter **Options** dialog window. When pasting a class to a package (no matter where this class was located before), its new instance appears in the namespace declaration corresponding to the package name. The namespace name will be the same as the package name, and files placed in the directory/package structure will have the corresponding namespaces in source code.

Warning: You cannot move packages because namespace organization will be violated. To put a package in a different location, you must create a new package with the same name located in that new location, create the necessary classes in it, then remove the old package.

Note: This example assumes the following:

You have already installed and activated Together support.

You have been working with your C# projects in Together ControlCenter 6.2 or Together Architect with the packaging namespace organization activated (see below) so that your source files are reflected in the appropriate namespaces in Developer Studio 2006.

To import a project, follow these steps:

1. Activating support for the packaging namespace organization
2. Setting up the project in Developer Studio 2006
3. Adding the Together ControlCenter file structure to the Developer Studio 2006 project
4. Viewing the Together ControlCenter project files in Developer Studio 2006

To configure TCC or TAR to automatically set namespaces for classifiers in implementation projects:

1. Open your project in TCC or TAR.
2. Choose **Tools** ▸ **Options** ▸ **Default level** or **Tools** ▸ **Options** ▸ **Project level**.
3. In the left pane of the **Options** dialog window, expand the **Source code** ▸ **C#** node of the options tree view.
4. Check **Set namespace for classes according to package name**.
5. Click **OK** to apply the settings and close the dialog box.
6. Close the TCC (TAR) project.

To set up a project in Delphi:

1. Choose **File** ▸ **New** ▸ **Other**. The **New Project** dialog box opens.
2. From the project types, choose C# Projects .
3. From the templates, choose the appropriate one.
4. Click **OK**.
5. Enter the project name. Use the same name as your TCC (TAR) project.

6 Choose a location for your project. The actual location for the project is irrelevant, but you will need to remember the location.

7 Click **OK**.

Result: The project is created and displayed in Developer Studio 2006.

To create the file structure:

1 Close the project in Developer Studio 2006.

2 Open Windows Explorer or any other file manager.

3 Navigate to your TCC or TAR project folder.

4 Copy the `src` folder from the TCC or TAR project folder to the Developer Studio 2006 project folder.

5 Open the `diagrams` folder in the TCC or TAR project and copy its contents to the `ModelSupport_%PROJECTNAME%ModelSupport` folder in the Developer Studio 2006 project folder. Developer Studio 2006 uses the `ModelSupport_%PROJECTNAME%ModelSupport` folder to save diagram files, TCC (TAR) uses the `diagrams` folder by default.

To add the TCC or TAR source code items to the new project:

1 Switch to Developer Studio 2006.

2 From the main menu, choose **File** ▸ **Reopen** and select the newly created project from the list.

3 Open the **Project Manager**.

4 Choose the project root node.

5 Right-click and choose **Add...** on the context menu. The **Add to Project** dialog box opens.

6 In this dialog box, choose the first source file from the `src` folder and click **OK**.

7 Repeat the last steps for all source and modeling files.

8 Open the **Model View**.

Result: The **Model View** displays the TCC or TAR diagram and source files.

Importing a Project Created in TVS, TEC, TJB, or TPT

Together supports full backward compatibility with the previous version. You can open your old projects in the regular way.

You can also import projects created in other editions of Together.

Warning: Diagrams in projects must be created in the common diagram format `.txv*`. The legacy diagram format `.df*` is not supported.

Warning: Diagram elements must be embedded (created as filemates). Standalone design elements (SDE) are not supported.

The general procedure for importing a project created in TVS, TEC, TJB, or TPT consists of the following steps:

1. Creating a new project in Developer Studio 2006
2. Importing the model information into this project

To create a new project for import:

1. Choose **File** ▸ **New** ▸ **Other** on the main menu. The **New Project** dialog window opens.
2. Select the project template. Note that the project type should correspond to the type of the source project:
   - For a C# project, choose **C# Projects** ▸ **(appropriate template)**.
   - For a UML 1.x design project, choose **Design project** ▸ **UML 1.5 Design Project**.
   - For a UML 2.x design project, choose **Design project** ▸ **UML 2.0 Design Project**.
3. Enter the project name.

   Warning: The project name should be exactly equal to the source project name. Adjust the remainder of the settings on your own.
4. Click **OK** to create a project.
5. Close the project when it is created.

To import the model information:

1. Open Windows Explorer or any other file manager.
2. Copy all model files including subfolders from the source project to the `ModelSupport_%PROJECTNAME%ModelSupport` folder under your new project root. These files are located under `diagrams`, `ModelSupport` or `Model Folder` directories, depending on the version of Together.

   Note: For some projects these files are located in the same folders as the source code files. In this case you will have to pick out the modeling files manually. Basically, you need all files with `.txv*` and `.txa*` extensions.
3. If you have an implementation project and you need to keep your source code, copy it from the source project to the new one keeping the folder structure.

4 Open the project in Developer Studio 2006. Open the **Project Manager**.

5 Choose the project root node.

6 Right-click and choose **Add...** on the context menu. The **Add to Project** dialog box opens.

7 In this dialog box, choose the first source file from the `src` folder and click **OK**.

8 Repeat the last steps for all source and modeling files.

Result: Developer Studio 2006 processes your files. When completed, the imported project is displayed in the **Model** and **Diagram Views**.

Importing a Project in IBM Rational Rose (MDL) Format

To create a design project on the base of an IBM Rational Rose (MDL) project:

1. On the main menu, choose **File** ▸ **New** ▸ **Other**. The **New Project** dialog box opens.
2. From the **Project Types** pane, choose Design Project.
3. From the **Templates** pane, choose **Convert from MDL** template.
4. Enter the project name, location and other parameters as required by the **New Project** dialog box.
5. Click **OK**.
6. In the **Project from MDL** wizard, specify the source `.mdl`, `.ptl`, `.cat`, or `.sub` file using the **Add** button.
7. Specify the scale factor and conversion options.
8. Click **Finish**.

Result: A new design project is created in the specified location.

Importing a Project in XMI Format

To import a project in XMI format:

1 Open a diagram or have the project root node selected in the **Model View**.

Warning: The project must comply with the UML 1.5 specification.

2 In the **Model View**, right-click the root project node and choose **Import Project from XMI**, or choose **File** ▸ **Import Project from XMI** on the main menu. The **XMI Import** dialog box opens.

3 Browse for the source file.

4 Click **Import**.

Tip: The recommended way to import a project from Together ControlCenter (TCC) or Together Architect (TAR) to Developer Studio 2006 is to use the common diagram format.

You can import a model created with IBM Rational Rose directly.

Opening an Existing Project for Modeling

You can add modeling capabilities to an existing implementation project that was created without Together.

When you open a project subdirectory from the **Model View** or **Diagram View**, Together reverse engineers the contents into a namespace diagram that shows the namespaces, classes, and interfaces and their interrelationships.

To open an existing implementation project for modeling:

1. Make sure that Together support is activated.
2. On the main menu, choose **File** ▸ **Open Project**.
3. In the **Open Project** dialog box, specify the project location.
4. Select the project or project group file.
5. Click **OK**.

Result: With Together support activated, opening existing implementation project automatically reverse engineers the existing source code into class diagrams.

Sharing a Project Between TCC/TAR and Developer Studio 2006

This section focuses on sharing your model information between Developer Studio 2006 and Borland Together ControlCenter (TCC) or Borland Together Architect (TAR) by using a C# project. You will create a set of diagrams in Together ControlCenter and then refer to these diagrams in Developer Studio 2006.

Use the following general procedure for creating a shared project:

1 Setting up a C# project in Developer Studio 2006

To set up a C# project:

1 On the main menu, choose **File** ▸ **New** ▸ **Other**. The **New Project** dialog box opens.
2 From the project types, choose C# Projects .
3 From the templates, choose the appropriate one.
4 Click **OK**.
5 Enter `ProjectRoot` as the project name.
6 Choose a location for your project. The actual location for the project is irrelevant, but you will create your Together ControlCenter project in the same location.
7 Click **OK**.

Result: The project is created and displayed in the **Project Manager**.

To create the folder hierarchy:

1 Navigate to the project folder by using Windows Explorer or any other file manager, and create a new folder under it.
2 Enter `src` for the name of the new folder.
3 Create another folder under `src`.
4 Enter `analysis` for the name of the new folder.
5 Repeat the last steps and add another folder naming it `requirements`.
6 Save all changes and close the Developer Studio 2006 project. If prompted to save changes to the project, click **Yes**.

The resulting folders created in Developer Studio 2006 will be used by TCC or TAR. The folder hierarchy begins with a `src` folder.

To create a project in Together ControlCenter or Together Architect:

1 Run TCC or TAR.
2 A corresponding TCC or TAR project must be created to share the diagram files. Choose **File** ▸ **New Project Expert** on the main menu.
3 On the first window of the **New Project Expert**:

- Specify the project name. For this example, enter `TCC_Project`.
- Specify the location of the project to match that of your project in Developer Studio 2006, `ProjectRoot`. For example, the Developer Studio 2006 project, `ProjectRoot`, was created at the following location: `C:\Documents and Settings\User\My Documents\Borland Studio Projects\ProjectRoot C:\Program Files\Microsoft Visual Studio .NET 2003\VC#\MyCSharpProjects`

\ProjectRoot. The TCC or TAR project location should be the same as the Developer Studio 2006 project location.

- Choose C# for the Default language.
- Choose **New project** for the project Creation Scenario.

4 Click **Next** to continue.

5 Select a path for the C# source files. Choose the src folder. For example, C:\Documents and Settings\User\My Documents\Borland Studio Projects\ProjectRoot\src C:\Program Files\Microsoft Visual Studio .NET 2003\VC#\MyCSharpProjects\ProjectRoot\src, and click **Next** to continue.

6 Select No to separate diagram files. Unlike Together ControlCenter, Developer Studio 2006 enforces separation between diagram files and source code.

7 Click **Next** to continue.

8 Select the ModelSupport_%PROJECTNAME% ModelSupport folder of the project root directory for the location to store the diagram files. For example, C:\Documents and Settings\User\My Documents\Borland Studio Projects\ProjectRoot\ModelSupport_ProjectRoot C:\Program Files\Microsoft Visual Studio .NET 2003\VC#\MyCSharpProjects\ProjectRoot\ModelSupport.

9 Click **Finish**.

Result: The Model tab of the Explorer pane displays the project structure, and the Designer pane displays the two project directories, analysis and requirements.

To configure Together ControlCenter to automatically set namespaces for classifiers:

1 Choose **Tools** ▸ **Options** ▸ **Default level** or **Tools** ▸ **Options** ▸ **Project level**.

2 In the left pane of the **Options** dialog window, expand the **Source code** ▸ **(language)** node of the options tree view.

3 Check **Set namespace for classes according to package name**.

Note: To make it possible to share models between TCC (TAR) and Developer Studio 2006, TCC (TAR) supports the packaging namespace organization for C# projects. This means that you can optionally require automatic settings of namespaces for C# classifiers in the **Options** dialog window in TCC (TAR).

When pasting a class into a package, no matter where this class was located before, its new instance appears in the namespace declaration corresponding to the package name. The namespace name will be the same as the package name, and files placed in the directory/package structure will have the corresponding namespaces in code.

4 Click **OK** to apply the settings and close the dialog box.

Warning: You cannot move packages. To move a package without violating namespace organization, create a new package with the same name located in the new location, create the necessary classes in it, then remove the old package.

To populate the analysis model:

1 In Together ControlCenter or Together Architect, double-click the analysis package in the Designer pane to open the analysis diagram.

2 Using the toolbar on the left side of the Designer pane, click the **Class** button.

3 Click once in the Designer pane to add the class to the diagram. Accept the default name for the class, `Class1`.

4 Repeat step 3, and add another class to the diagram. Accept the default name for the class, `Class2`.

To populate the requirements model:

1 In Together ControlCenter or Together Architect, open the Model tab of the Explorer pane, double click the requirements diagram to open it in the Designer pane.

2 Click the **New Diagram** button in the Designer pane toolbar. The **New Diagram** dialog box opens.

3 Select the Use Case diagram in the **New Diagram** dialog box.

4 Enter `PlaceOrderUseCase` in the Diagram name field.

5 Click **OK**. The `PlaceOrderUseCase` diagram opens in the Designer pane.

6 Using the toolbar on the left side of the Designer pane, click the **Actor** button.

7 Click once in the Designer pane to add an actor to the diagram. Accept the default name for the actor, `Actor1`.

8 Click the **Use Case** button.

9 Click once in the Designer pane to add the use case to the diagram. Accept the default name for the use case, `UseCase1`.

Result: The diagrams are now complete and ready for use in Developer Studio 2006. The Model tab of the Explorer pane displays the project structure.

To access the diagrams created with Together ControlCenter or Together Architect:

1 Save your project and close Together ControlCenter or Together Architect.

2 Switch to Developer Studio 2006.

3 From the main menu, choose **File** ▸ **Recent Projects**, and select the `ProjectRoot` from the list.

4 Add the newly added model elements by using the **Project Manager**.

5 The source files, `Class1.cs` and `Class2.cs`, were added while working in Together ControlCenter or Together Architect.

6 Expand the `src` and `analysis` folders in the **Project Manager**. The `Class1` and `Class2` nodes are present.

The **Model View** updates and reflects the `Class1` and `Class2` source files in the analysis diagram. Now, changes made to your diagrams in Developer Studio 2006 will appear on the diagrams when working in TCC or TAR.

Warning: When adding a source-generating element (such as a class or interface) to a namespace through a class diagram in Together, Developer Studio 2006 physically adds the source-generating element to the project root. Together does not control where Developer Studio 2006 places its source code files; however, Together will display them correctly on the class diagram. Use the **Project Manager** and drag-and-drop the source-generating element to the proper folder, so that when you work with the project in Together ControlCenter or Together Architect, the source contents appear in the correct location.

Synchronizing the Model View, Diagram View, and Source Code

Together provides constant synchronization between different aspects of your project:

- Model hierarchy, presented in the **Model View**
- Model graphical representation in the **Diagram View**
- Source code (for implementation projects)

Tip: You can also use the Reload function of the **Model View** to update an entire model, and the Refresh function of the **Diagram View**.

You can navigate between the Model View, Diagram View, and source code by using the following techniques:

1. Navigate to a diagram from the **Model View** to the **Diagram View**
2. Navigate to a model element from the **Model View** to the **Diagram View**
3. Navigate from the **Diagram View** to the **Model View**
4. Navigate from a lifeline to its classifier in the **Model View** or a Class diagram
5. Navigate from source code to the **Model View**
6. Navigate from the **Model View** or **Diagram View** to source code (for implementation projects)
7. Edit a synchronized element

To navigate to a diagram from the Model View to the Diagram View:

1. In the **Model View**, right-click the diagram node.
2. Choose **Open Diagram**.

Alternatively, double-click the diagram node in the **Model View**.

To navigate to a model element from the Model View to the Diagram View:

1. Select a model element in the **Model View**.
2. Right-click and choose **Select on Diagram** on the context menu.

Note: If this model element appears on several diagrams, choose a diagram on the submenu.

To navigate from the Diagram View to the Model View:

1. Right-click the selected element or diagram background in the **Diagram View**.
2. Choose **Synchronize with Model View** on the context menu.

To navigate from a lifeline to its classifier in the Model View or a Class diagram:

1. Right-click the selected lifeline on a UML 2.0 Sequence diagram in the **Diagram View**.
2. Choose **Select** ▸ **Type in Model View** to navigate to the classifier in the **Model View**,

   Or:

Choose **Select** ▸ **Type on Diagram** to navigate to the classifier on a Class diagram in the **Diagram View**.

To navigate from source code to the Model View:

1. Right-click the line that contains the desired element.
2. On the context menu of the selection, choose **Synchronize Model View**.

Result: The corresponding element is highlighted in the **Model View**.

To navigate from the Model View or Diagram View to source code (for implementation projects):

1. Right-click a model element or a node member.
2. Choose **Go to definition** on the context menu.

Note: This command is available for source code-generating elements.

Result: Source code of the element in question opens in the Editor tab. The corresponding definition is highlighted.

Tip: To open source code of an entire class or interface, double-click the element icon.

To edit a synchronized element:

1. Select an element in the **Diagram View** or **Model View**.
2. Edit the desired fields in the **Object Inspector**.

Note: Alternatively, invoke the in-line editor in the **Diagram View** or **Model View**.

Warning: Avoid using the **Structure View** or the **Project Manager** for modification of the model elements.

Transforming a Design Project to Source Code

This feature is available for UML 1.5 and UML 2.0 design projects.

To generate source code from a design project:

1. In the **Model View**, select a design project.
2. Right-click and choose **Transform to source** on the context menu.
3. In the **Choose Destination Project** dialog box, select the desired implementation project.
4. Check the **Use name mapping files for code generation** checkbox if required.
5. Click **Transform**.

Result: implementation code of the class diagrams that existed in the design project are added to the target language-specific project. The diagrams are also added to the target project. The diagram roots are preserved.

To insert source code to an implementation project:

1. In the **Model View**, select an implementation project.
2. Right-click and choose **Transform code from design project** on the context menu.
3. In the **Choose Source Project** dialog box, select the desired design project.
4. Check the **Use name mapping files for code generation** checkbox if required.
5. Click **Transform**.

Result: implementation code of the class diagrams that existed in the design project are added to the target implementation project. The diagrams are also added to the target project. The diagram roots are preserved.

Troubleshooting a Model

You can also reload your project from the source code.

Use the following techniques to troubleshoot your model:

1. Refresh a model
2. Reload a model
3. Fix a model

To refresh a model:

1. Open the **Diagram View**.
2. Press F6.

To reload a model:

1. Open the **Model View**.
2. Right-click the project root node and choose **Reload** on the context menu.

Note: Use the Reload command as a workaround for issues that might appear while making changes in Together that cause some elements on the diagram to stop responding, or if you get errors from Together, such as, `<undefined value>`.

Tip: Usually, when these problems occur, the elements also disappear from the Developer Studio 2006**Structure View** and the corresponding source code is underlined in blue in the Developer Studio 2006 Editor. Together cannot always properly handle such elements that become broken. To restore broken elements to a normal state, it is necessary to edit the code in the text editor according to the recommendation shown in the Developer Studio 2006 Editor. In these cases, it is best to refresh the model using Reload to prevent possible further misbehavior.

To fix a model:

1. For interaction diagrams: regenerate them from the source code.
2. For all types of diagrams: check that none of the necessary elements are hidden.

Working with a Namespace or a Package

Namespaces are used in implementation projects, and packages in design projects.

Use the following techniques for a namespace or a package:

1. View a namespace or a package
2. Open a namespace or a package
3. Delete a namespace or a package
4. Rename a namespace or a package

To view a namespace or a package:

1. By default, a namespace element on a diagram displays the namespace contents.
2. You can use the context menu of a class or interface in a namespace to add fields and methods directly.

To open a namespace or a package:

1. Choose the **Open Diagram** command on the namespace diagram context menu.
2. You can also double-click the namespace element on the diagram.

To delete a namespace or a package:

1. Open the **Diagram View** or the **Model View**.
2. Choose **Delete** on its context menu.

> **Warning:** Deleting a namespace also deletes all of its contents.

To rename a namespace or a package:

1. Open a project.
2. To rename a namespace, including changing the namespace name in all of its source files, do one of the following:
   - Choose **Rename** on the context menu of a namespace in the **Diagram View** or in the **Model View**
   - Invoke the in-place editor for the namespace element in the **Diagram View** or in the **Model View**
   - Edit the Name field in the **Object Inspector**

Working with a Referenced Project

Your project can have a binary library whose content you may want to display in your diagrams. For example, you can show entities that reside in the `MSCorLib.dll` or other project references. Such resources exist for the project, but Together does not include them in the generated HTML documentation for the project.

The **Model View** enables you to view class diagrams for references included in your projects. You can add references to your project using the **Project Manager**.

To add a project to references:

1. In the **Project Manager**, expand the desired project node.
2. On the context menu of the References node, choose **Add Reference**.

   Tip: Alternatively, choose **Project** ▸ **Add Reference** on the main menu.

3. In the **Projects** tab, select the projects to be referenced and click **Select**.
4. Click **OK** when ready.

Result: The **Choose Type to Instantiate** dialog box shows all referenced projects, making it possible to choose the desired classifiers from the different projects.

To view a diagram of a referenced project:

1. Open or create a class diagram.
2. Right-click the diagram background and choose **Add** ▸ **Shortcuts**. The **Edit Shortcuts** dialog box opens and displays the content available for the diagram and all content residing outside of the current namespace.
3. Choose the resource that you want to add from the tree view of available contents on the left of the dialog and click **Add >>**.
4. Repeat until you have added all of the resources that you want to show on the diagram.
5. Click **OK** to close the dialog box.

Tip: If the **Edit Shortcuts** dialog box does not show the resource that you are looking for, it is probably not added as a reference to your project. Choose **Project** ▸ **Add Reference** on the main menu to add a project reference.

To view the MsCorLib.dll (a standard DLL added automatically to your projects):

1. Expand the **References** node and the `MsCorLib.dll` node in the **Model View**.
2. Right-click the default diagram and choose **Open Diagram**.

   The default diagram opens in the **Diagram View**. You can expand the Microsoft and System folders to view other class diagrams as well.

Together Quality Assurance Procedures

Creating a Metrics Chart

You can create a chart in the **Metric Results Pane**.

Metrics charts are created in temporary files which are deleted when the charts are closed. However, you can save graphical information in text files, export it to the desired graphical format, and include graphics in project.

To create a bar chart:

1. Select a column that contains the result for the desired metric.
2. Right-click and choose Bar Chart.

To create a Kiviat chart:

1. Select the row that contains the results for the desired element.
2. Right-click and choose Kiviat Chart.

To save a chart:

1. Right-click the chart tab and choose Save.
2. In the Save graph dialog box, navigate to the target location and click Save.

To export a chart to image:

1. Select the desired chart.
2. On the main menu, choose File | Export diagram to image.
3. In the Export diagram to image dialog, specify the zoom factor and image dimensions.
4. Click Save.

To add a chart to project:

1. Select the desired chart.
2. On the main menu, choose File | Move [chart name] to Project.
3. On the submenu, select a project within the current project group.

Exporting Audit Results

Export audit results to an XML or HTML file to share them with team members or review them later.

To save the audit results in a separate file:

1. Select the rows of the table that you want to save. Do not select anything if you want to print the entire list.
2. Click the **Save** button on the toolbar.
3. In the **Save Audit Results** dialog box that opens, choose the scope of the results to export using the **Select View** list box:

   - **All Results**: If the results are grouped, choosing All Results prints a report for all groups in the current tabbed page. If the results are not grouped, then all results export for the current tabbed page.
   - **Active Group**:If the results are grouped, you can select a group in the current tabbed page, and the generated report contains the results from the selected group.
   - **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing Selected Rows generates a report for such selections.

4. Each tabbed page can contain a list of audits (when the audits are ungrouped) or a group tree with a list of the selected group (when the audits are grouped).

   Note: Unless the results have been grouped using the Group by command, the Active Group option is not enabled in the dialog box.

   Tip: You can use CTRL+CLICK to select multiple rows.

5. In the Select Format list box, select the format for the exported file:

   - **XML**: Generates an XML-based report.
   - **HTML**: Generates an HTML-based report.

   Selecting HTML format activates the following check boxes:

   - **Add Description**: This saves the audit descriptions in a separate folder with hyperlinks to the descriptions from the results file.
   - **Launch Browser**: This option opens the generated HTML file in the default viewer.

6. Click **Save** to save the results in the specified location.

Printing Audit Results

You can print the entire table of audit violations, or select specific rows and columns.

Warning: This feature is available for implementation projects only.

To print the list of audit violations:

1 Select the rows of the table that you want to print. Do not select anything if you want to print the entire list.

> **Tip:** You can select multiple rows using CTRL+CLICK.

2 Click the Print button on the Toolbar. The Print Audit dialog box opens.

3 Choose the scope of the results to print using the Select View list box:

- **All Results**: If the results are grouped, choosing All Results prints a report for all groups in the current tabbed page. If the results are not grouped, then all results print for the current tabbed page.
- **Active Group**: If the results are grouped, you can select a group in the current tabbed page, and the printed report contains the results from the selected group.
- **Selected Rows**: You can select single or multiple rows in the audit results report view. Choosing Selected Rows prints a report for such selections.

4 Each tabbed page can contain a list of audits (when the audits are ungrouped) or a group tree with a list of the selected group (when the audits are grouped).

> **Note:** Unless the results have been grouped using the Group by command, the Active Group option is not enabled in the dialog window.

5 If desired, specify the print zoom factor in the Print zoom field, or check Fit to page if you want to print the results on a single page. If Fit to page is checked, the Print zoom field is disabled.

6 If necessary, adjust the page and printer settings:

- Click the Print list box, and choose the Print dialog box command to select the target printer.
- Choose **Tools** ▸ **Options** and open **Together** ▸ **(level)** ▸ **Diagram** ▸ **Print** options to set up the paper size, orientation, and margins.

> **Tip:** Click the drop-down arrow to the right of the Preview option to open the preview pane. Use the Preview zoom (auto) slider, or Auto preview zoom check box as required. Click the upward arrow to the right of the Preview option to close the preview pane.

7 Click Print to open the system print dialog box, and send the file to the printer.

Running Audits

Audits automatically check for conformance to standard or user-defined style, maintenance, and robustness guidelines. Before running audits, make sure that the code being audited is compilable. If your source code contains errors, or some libraries and paths are not included, audits might produce inaccurate results.

Warning: This feature is available for implementation projects only.

To run audits:

1. Open an implementation project.
2. Open the **Model View**.
3. Right-click the project root node. **QA Audits** on the context menu. The **Audits** dialog window opens.
4. In this dialog window:
   - In the Scope list box, choose the code to run the set of audits on.
   - Model processes the entire project.
   - Selection processes only the specific classes, namespaces, or diagrams currently selected in the Diagram or Model View.

   Tip: If you have not selected any items in the Diagram or Model View, the Scope option defaults to the entire project.
5. If you want to run audits on specific classes, namespaces, or diagrams, make sure you correctly select them before you open the Audits dialog window.
6. Choose the audits to run. As you click an audit, the description for each audit is shown in the lower pane of the dialog box.
7. For each audit, the severity level and other audit-specific options are displayed in the right-hand panel of the Audits dialog box. Change the settings if necessary.
8. When you have selected your set of audits, click Start. The **Operation in progress** dialog box opens displaying a status bar that indicates the progress completed. The status bar will display until the process finishes.
9. If necessary, click Cancel to abort the process.

   Note: Audits run in the command thread, so you cannot edit the project while they are being processed.

The Audits Results Pane opens automatically, displaying the results. In the results table, right-click any line to open the context menu and use its commands to perform operations with the report.

Running Metrics

Before running metrics, make sure that the code being analyzed can be compiled. If your source code contains errors or some libraries and paths are not included, metrics might produce inaccurate results.

Warning: This feature is available for implementation projects only.

To run metrics:

1. Open an implementation project.
2. Open the **Model View**.
3. Right-click the project root node. **QA Metrics** on the context menu. The **Metrics** dialog window opens.
4. In this dialog window:
   - In Scope, choose what to run metrics on: Model processes the entire project.
   - Selection processes only the specific classes, packages, or diagrams currently selected in the diagram or Model View.
5. Choose the metrics you want to analyze. Each metric displays a description in the lower panel of the Metrics dialog box.

   Tip: If nothing is currently selected in the diagram or navigator view, the Selection scope is not available. If you want to run metrics on specific classes, packages, or diagrams, make sure you correctly select them before you open the Metrics dialog window.

6. For each metric there are settings for options such as limits and granularity in the right-hand panel of the Metrics dialog box. Change the settings if necessary.
7. When you have selected your set of metrics, click Start.

   Note: Metrics run in the command thread, so you cannot edit the project while they are being processed.

Result: The Metrics Results Pane opens automatically displaying the results.

Viewing Audit Results

When viewing audit results, you can compare and organize items in the results report.

The results report is tightly connected with the diagram elements and the source code. Using the report, you can navigate to the specific location of the violation.

Warning: This feature is available for implementation projects only.

Use the following techniques when viewing audit results:

1. Sort all the items according to the values for a specific column
2. Group items according to the current column
3. Navigate to the specific location of the violation

To sort all the items according to the values for a specific column:

1. Switch to the audit results table.
2. Click the column heading. The arrow in the heading displays whether sorting is ascending or descending.

To group items according to the current column:

1. Right-click the Audit results table and choose Group By. This enables you to organize the results by changing the relationship of rows and columns.
2. To ungroup the results, right-click the table, and choose Ungroup.

To navigate to the specific location of the violation:

1. Select any element in the results report.
2. Choose Open on the context menu (or just double click the row) to navigate directly to the source code.

Viewing Metric Results

Use the following techniques when viewing metric results:

1. Sort results by column
2. Filter results
3. Update results
4. Navigate to the source code
5. View the metric description

To sort results by column:

1. Select the desired column in the metrics result table.
2. Click the column header to change the sorting order.

To filter results:

1. You can filter the displayed results to improve the meaningfulness of the results report.
2. Use the following toolbar buttons to show and hide elements:

Button
Namespaces
Classes
Methods
Child elements

To update results:

1. You can update or refresh the results table.
2. Use the following **Tool Palette** buttons:

Button	Description
Refresh	Recalculate the results that are currently displayed
Restart	Open the Metrics dialog window, define new settings and start new metrics analysis.

To navigate to the source code:

1. Select the row in the results table that is of interest to you
2. Right-click and choose **Open** on the context menu to navigate directly to it in the source code.

To view the metric description:

1. Select the column in the results table that corresponds to the metrics of interest to you.
2. Right-click and choose **Show description** on the context menu.

Working with a Set of Audits

To create a set of audits:

1. On the main menu choose **Tools** ▸ **Together** ▸ **QA Audits**. The dialog window **QA Audits** opens.
2. Toolbar buttons in the dialog window provide commands for working with the sets of audits.
3. If you want to base your new saved set on the default set, click the **Set default audit set** button.
4. If you want to base it on a previously created custom set, click the **Load set** button, then choose the desired saved `.adt` file.
5. Go through the individual audits and check those you want to include in the set, or clear those you do not want to include.
6. Select all the items in a group by checking the group name.
7. When you complete your selection, click the **Save set** button, and specify the location and filename for new set file.

To use a saved set of audits:

1. On the main menu choose **Tools** ▸ **Together** ▸ **QA Audits**. The dialog window **QA Audits** opens.
2. Click the **Load Set** button and choose the `.adt` file you want to use.
3. Click **Start**.

Tip: You might want to include the `.adt` files in your backup routine.

Working with a Set of Metrics

To create a set of metrics:

1. On the main menu choose **Tools** ▸ **Together** ▸ **QA Metrics**. The dialog window **QA Metrics** opens.
2. Toolbar buttons in the dialog window provide commands for working with the sets of metrics.
3. If you want to base your new saved set on the default set, click the **Set default metric set** button.
4. If you want to base it on a previously created custom set, click the **Load set** button, then choose the desired saved `.mts` file.
5. Go through the individual metrics and check those you want to include in the set, or clear those you do not want to include.
6. Select all the items in a group by checking the group name.
7. When you complete your selection, click the **Save set** button, and specify the location and filename for new set file.

To use a saved set of metrics:

1. On the main menu choose **Tools** ▸ **Together** ▸ **QA Metrics**. The dialog window **QA Metrics** opens.
2. Click the **Load set** button and choose the `.mts` file you want to use.
3. Click **Start**.

Tip: You might want to include the `.mts` files in your backup routine.

Together Refactoring Procedures

Refactoring: "Safe Delete"

To safely delete an element:

1 Select the element to be deleted.

2 On the main menu, choose **Refactoring** ▸ **Safe Delete**

> **Tip:** Alternatively, right-click on the element and choose **Refactoring** ▸ **Safe Delete** on the element's context menu.

3 In the **Safe Delete** dialog box that reports the element to delete and any usages of that element:

- If no usages are found, press **Delete**.
- If usages are found, click **View usages**. The Refactoring window opens allowing you to review the refactoring before committing to it. Click the **Perform refactoring** button to delete the element.

Refactoring: Changing Parameters

To change parameters, follow these steps:

1 Select method in the **Diagram View**, in the **Model View** or in the Editor.

2 Choose **Refactoring Change Parameters** from the main menu.

> **Tip:** Alternatively, you can right-click and choose **Refactoring Change Parameters** on the context menu.

3 In the resulting dialog, select parameter from the list and choose the desired action:

- To add a new parameter, click **Add**, and specify the parameter name, type and default value.
- To delete parameter, click **Remove**.
- To rename parameter, click the **Name** field, and edit the parameter name using the in-place editor.

4 If applicable, check **Refactor Ancestors.**

5 Check **Preview Usages** if necessary.

- If this option is checked when you click OK, the **Refactoring** window opens allowing you to review the refactoring before committing to it. Click the **Perform refactoring** button to complete the changes. You can use the **Undo** and **Redo** commands as necessary once you have performed the refactoring.
- If this option is cleared when you click OK, the **Refactoring** window opens with the change completed. You can use the **Undo** and **Redo** commands as necessary once you have performed the refactoring.

Refactoring: Creating Inline Variables

To create an inline variable:

1. Select the local variable in the Editor.
2. On the main menu, chooseRefactoring ▸ Inline variable

   Tip: Alternatively, you can choose Refactoring ▸ Inline variable on the context menu.

   The resulting dialog reports the number of variable occurrences that the Inline Variable command will be applied to.
3. Click OK to complete refactoring.

Warning: The variable that you select for creating an inline variable, should not be updated later in the source code. If it is, the following error message will display: "Variable index is accessed for writing."

For example, if you use the Inline Variable refactoring command on the local variable, index, shown below:

```
public void findIndex() {
                        int index = 2;
                        System.Console.Writeline("Index is: {0}", index);
                }
```

then the following refactoring occurs:

```
public void findIndex() {
   System.Console.Writeline("Index is: {0}", 2);
}
```

Refactoring: Extracting Interfaces

The following conditions should be met for extracting interfaces:

- Only non-static methods can be extracted.
- All methods in the extracted interface are public.
- If the name specified for the new interface coincides with the name of an existing interface in the same namespace, all the methods will be extracted into an existing interface.

To extract an interface:

1. Select one or more code elements (class, interface, field, method, event, property, or indexer) in the Diagram or **Model View**.
2. On the main menu, choose**Refactoring** ▸ **Extract superclass**

   Tip: Alternatively, you can choose **Refactoring** ▸ **Extract superclass** on the context menu of the selection.

3. In the **Extract interface** dialog, enter the name for the interface and designate its namespace, if applicable.
4. Specify the members to be used in the resulting superclass or interface by setting or clearing the respective check-boxes.
5. Click **OK**. The **Refactoring** window opens allowing you to review the refactoring before committing to it.
6. Click the **Perform refactoring** button to complete the extraction.

Refactoring: Extracting Method

To extract a method:

1 In the Editor, open the class or interface containing the code fragment that you wish to extract.

2 Place the mouse cursor in the desired fragment of source code. Refactoring determines the beginning and the end of the relevant statement.

3 On the main menu, choose **Refactoring** ▸ **Extract Method**

> **Tip:** Alternatively, right-click the code fragment and choose **Refactoring** ▸ **Extract Method** on the context menu.

4 In the dialog box that opens, specify the following information:

- Name of the new method
- Visibility (public, protected, private, internal, internal protected)
- Header comment
- Whether the method is Static.

5 Click **OK** to complete the extraction and create the new method.

- When applying Extract Method, parameters and local variables in the selected code fragment become the parameters of the new method.
- The code fragment cannot contain a return statement of the original method. An error message displays if you attempt to include a return statement in the code fragment.
- The code fragment cannot modify more than one single local variable. An error message displays if you violate this restriction.
- If the selected code fragment is repeated in several locations, it is your responsibility to replace these fragments in the appropriate locations with the proper method calls.

Refactoring: Extracting Superclass

To use the "Extract superclass" operation:

1. Select one or more code elements (class, interface, field, method, event, property, or indexer) in the Diagram or **Model View**.
2. On the main menu, choose**Refactoring** ▸ **Extract superclass**

   Tip: Alternatively, you can choose **Refactoring** ▸ **Extract superclass** on the context menu of the selection.

3. In the **Extract superclass** dialog, enter the name for the interface and designate its namespace, if applicable.
4. Specify the members to be used in the resulting superclass or interface by setting or clearing the respective check-boxes. If applicable, indicate that a method is abstract in the extracted superclass.
5. Click **OK**. The **Refactoring** window opens allowing you to review the refactoring before committing to it.
6. Click the **Perform refactoring** button to complete the extraction.

Refactoring: Introducing Fields

To introduce a field:

1 Select expression in the Editor.

2 On the main menu, choose**Refactoring** ▸ **Introduce Field**

> **Tip:** Alternatively, you can choose **Refactoring** ▸ **Introduce Field** on the context menu.

3 In the resulting dialog, specify the following:

- **Name:** Enter the name of the new field
- **Visibility**: Using the list box, choose the visibility for the new field from *public, protected, private, internal,* or *internal protected*.
- **Initialize**: Choose where to initialize the new field. Using the list box, choose from *Current method, Class constructor(s)*, or *Field declaration*.

4 If applicable, check the Static and Replace all occurrences fields.

5 Click **OK** to complete the refactoring.

Refactoring: Introducing Variables

To introduce a new variable:

1. Select variable in the Editor.
2. On the main menu, choose**Refactoring** ▸ **Introduce Variable**

   Tip: Alternatively, you can choose **Refactoring** ▸ **Introduce Variable** on the context menu.

3. In the resulting dialog, specify the Name of the new variable. The new variable created is given the same type as the original variable.
4. If desired, check **Replace all occurrences**. The **Introduce Variable** dialog indicates the number of occurrences that it will replace with the new variable.

   Note: The refactoring does not replace any occurrences of the variable prior to the point in the code at which you selected to introduce the new variable.

Refactoring: Moving Members

To move a static member to a different class:

1. Select one or more static members in the **Diagram View** or **Model View**.
2. On the main menu choose **Refactoring** ▸ **Move**

   Tip: Alternatively, right-click on the selection and choose **Refactoring** ▸ **Move Members** on the context menu

3. In the **Move Members** dialog, use the **Move Members** field to select which static members to move. You can deselect/select the static members by clearing/checking the check box next to the name of the member
4. Use the **To** field to enter the fully-qualified name for the target class where the selected code element or elements will reside.
5. Click **OK**.

Refactoring: “Pull Members Up" and “Push Members Down”

Moving members assumes that the member is either moved to the target location being deleted from the original location, or created in the target location being preserved on the original one.

To move a member:

1 Select member in the **Diagram View** or in the **Model View**.

Tip: In the editor, place the mouse cursor on the member name.

2 Choose **Refactoring** ▸ **Pull Members Up/Push Members Down** on the context menu or on the main menu.

3 In the resulting dialog box, specify additional information required to make the move.

- In the top pane of the dialog box, check the members to be moved.
- In the bottom pane of the dialog box, that shows the class hierarchy tree, select the target class.

4 Click **OK**.

5 In the **Refactoring** window that opens, review the refactoring before committing to it. Click the **Perform refactoring** button to complete the move.

Tip: Moving members is more complicated than moving classes among namespaces, because class members often contain references to each other. A warning message is issued when **Pull Members Up** or **Push Members Down** has the potential for corrupting the syntax if the member being moved references other class members. You can choose to move the class member and correct the resulting code manually.

Unit Test Procedures

Building Tests

The structure of a unit test is largely dependent on the functionality of the class and method you are testing. The Unit Test Wizards can help you by providing a template of the test project, setup and teardown methods, and basic tests. You will need to add the specific test logic to test a particular method. The following procedures describe how to build your test projects and test cases. Follow these procedures in order. The test project must be built prior to the test cases.

To build a test project

1. Choose **File** ▸ **New** ▸ **Other**.
2. Open the **Unit Test** folder.
3. Double-click the **Test Project** gallery item.

   This starts the **Test Project Wizard**.
4. Enter the project name or accept the default name.
5. Enter the location or accept the default location.
6. Select the personality or accept the default.

   By default, the personality is set to the same personality as the active project.
7. If you do not want the test project added to your project group, uncheck the **Add to Project Group** check box.
8. Click **Next**.
9. Choose the GUI or Console test runner, then click **Finish**.

   The **Test Project Wizard** adds the necessary references to your project.

To build a test case

1. Click the **Code** tab for the file containing the classes you want to test.

   This makes the file active in the **Code Editor**.
2. Choose **File** ▸ **New** ▸ **Other**.
3. Open the **Unit Test** folder.
4. Double-click the **Test Case** gallery item.

   This starts the **Test Case Wizard**.
5. Choose a source file from the **Source File** drop down list.

   All source files in your project are listed.
6. Select the classes and methods you want to build tests for, by checking or unchecking the check boxes next to the class and method names, in the **Available classes and methods** list.

   Note: You can deselect individual methods in the list. The wizard will build test templates for the checked methods only. If you deselect a class, the wizard will not create test templates for any of the methods in that class.
7. Click **Next**.

   This displays the next page of the **Test Case Wizard**.
8. Fill in the appropriate details or accept the defaults.
9. Click **Finish**.

The wizard creates a test case file and creates a name for the file by prefixing the name of the active code file with the word Test. For example, if your code file is named MyProgram, the test case file will be named TestMyProgram.

To write a test case

1. Add code to the SetUp and TearDown methods, if needed.
2. Add asserts to the test methods.

To run the test case in the GUI Test Runner

1. Click the **Code** tab for the file containing the classes you want to run.
2. Choose **Run** ▸ **Run**.

   The **GUI Test Runner** starts up immediately on execution of your application. The list of tests appears in the left pane of the **GUI Test Runner**.
3. Select one or more tests.
4. Click the **Run** button.

   The test results appear in the **Test Results** window. Any test highlighted with a green bar passed successfully. Any test highlighted in red failed. Any test highlighted in yellow was skipped.
5. Review the test results.
6. Fix the bugs and rerun the tests.

Concepts

.NET

Building Applications with the ECO framework

The integrated modeling tools in Developer Studio 2006 tie together the processes of design and development. The structure and behavioral modeling tools integrated into the IDE are based on industry standards such as the Universal Modeling Language (UML) and the Object Constraint Language (OCL). The Enterprise Core Object (ECO) framework leverages the .NET framework to make the model available at both designtime and runtime. This section provides an overview of the ECO framework, and introduces basic concepts needed to work with the framework.

In This Section

Overview of the ECO framework
Describes architecture of the ECO framework.

ECO Modeling Tools Overview
Describes ECO UML modeling tools available in the Developer Studio 2006 IDE.

Working with the ECO Service API
Describes how to access the services provided by the ECO framework.

Working with ECO Handles
Defines the concept of a handle, and describes how they are used in the ECO framework.

Modeling Behavior with State Machines
Describes simple UML state machines and how they are used with the ECO framework.

Using Substates with the ECO framework
Describes how to use composite states, substates and regions with ECO state machine diagrams.

Overview of the Object Constraint Language
Describes the Object Constraint Language (OCL) in a high-level overview.

Using ECO Action Language
Describes the extensions provided by the ECO Action Language.

Working with ECO Subscriptions
Describes how to work with the ECO subscription mechanism.

The ECO framework and ASP.NET
Describes basic concepts required for understanding how to build an ECO ASP.NET application.

Using the ECO Framework in Multi-Client Applications
Describes the concepts and components used when writing multi-client ECO framework applications.

Custom ECO Object-Relational Mapping Files
Describes the format of the object-relational mapping file used by the ECO framework.

Building Applications with the ECO Framework
Describes the core processes of building an application using the ECO framework.

Introduction

These topics contain introductory material on working with the ECO framework.

In This Section

Overview of the ECO framework
Describes architecture of the ECO framework.

ECO Modeling Tools Overview
Describes ECO UML modeling tools available in the Developer Studio 2006 IDE.

Working with the ECO Service API
Describes how to access the services provided by the ECO framework.

Working with ECO Handles
Defines the concept of a handle, and describes how they are used in the ECO framework.

Working with ECO Subscriptions
Describes how to work with the ECO subscription mechanism.

Overview of the ECO framework

This topic gives an overview of the designtime and runtime features of the ECO framework.

Introduction to the ECO framework

The ECO framework is an object-relational mapping framework. Object-relational mapping is a process that abstracts relational database concepts, and maps them to object-oriented programming concepts. In practical terms, the ECO framework maps relational database rows to C# or Delphi for .NET objects, thereby relieving the programmer from writing low-level SQL code.

To reduce the amount of manual coding even further, Developer Studio 2006 integrates the ECO framework with TogetherLiveSource UML diagram tools. This allows you to specify both structure and behavior visually, using UML class diagrams and UML state machine diagrams. Integration of UML designers with the ECO framework gives you the ability to work with familiar object-oriented concepts, while the framework handles the mapping and storing of objects in a relational database.

One unique feature of the ECO framework is that it adheres to the philosophy that models can be both *implemented* and *executed* . A precisely described model contains enough information that much of the source code needed to bring the model to life can be generated automatically, as opposed to being written by hand. This is the difference between automatic implementation of a visually constructed model, and interpretation (and coding by hand) of that model by a programmer.

Execution of the model means that the designtime support for creating the model carries through to runtime. A truly design driven software engineering process includes support not only for creation of a model, but also for maintaining and enforcing the integrity of the model at all phases of the application's lifetime.

ECO framework Terminology

The following list defines some important terminology that you will encounter throughout the ECO framework. These terms and their related concepts are covered in more detail in separate topics. Please refer to the links below for more information.

ECO space	An object store that contains objects created during the lifetime of the application. The ECO framework handles the mapping of object attributes, storage, and retrieval in a relational database. At runtime, the ECO Space contains all of the metadata of the model, *plus* the instances of the classes in your model. Think of the ECO Space as an instance of a model, much like an object is an instance of a class. The objects contained in the ECO Space retain the domain properties (attributes and operations) and relationships defined in the model.
Handle	An opaque reference that binds to an object, or a collection of objects in an ECO space.
Object Constraint Language (OCL)	A formal language used to query the ECO space. An OCL expression returns a single value, a single object, or a collection objects. An OCL expression cannot alter the value of any object attribute.
ECO Action Language	An extension of OCL that allows you to change the value of object attributes.
Service API	The most commonly accessed runtime functionality of the ECO framework is grouped into a namespace called `Borland.Eco.Services`. The `Borland.Eco.Services` namespace defines a number of interfaces that you will use to access and manipulate the objects in the ECO space. At runtime, you obtain these interfaces by accessing properties of the ECO space.

Designtime Functionality

It is helpful to understand how the ECO framework splits its functionality into designtime support and runtime support. The following table shows designtime features, and briefly describes how you work with the feature within the IDE.

Designtime Feature	IDE Interaction	Notes
Model-oriented source code navigation	**Model View**	Choose **View** ▸ **Model View** to open the **Model View**.
Object Store (ECO space) configuration	**ECO space designer**	All ECO projects contain an object store called an ECO space. When the ECO space source file is active in the **editor**, click the **design** tab to open the **ECO space designer**.
Model validation	**Validate model** tool on the **ECO space designer**	Click the **Validate model** tool button in the **ECO space designer** to perform model validation.
Structural design of the model	**ECO class diagram**	Create new diagrams using context menus in the **Model View**.
Behavioral design of the model	**ECO state machine diagram**	Create new diagrams using context menus in the **Model View**.
Object Constraint Language (OCL) expression editor	**ECO class diagram** **ECO state machine diagram** **ECO Winform designer**	Most ECO components have properties that are expressed using OCL. This includes ECO class attributes, association ends, and behavioral features modeled on state machine diagrams. The **OCL expression editor** is a property editor, opened from the **Object Inspector**. It allows you to easily build expressions to query and perform operations on the ECO space.
Reverse engineer an existing relational database, extracting classes and associations, and wrapping them in ECO source code	**Reverse and wrap database** tool on the **ECO space designer**	Click the tool button in the **ECO space designer** to start a wizard that configures the ECO space and generates ECO source code for an existing relational database. The reverse engineering wizard also handles the case where data is stored across multiple databases.
GUI design	**ECO Winform designer**	ECO framework applications use standard .NET controls to display and edit data.
Handle configuration	**ECO Winform designer**	Handles bind to objects in the ECO space using OCL expressions. In the **ECO Winform designer**, drop a handle component on the form and configure it using the **OCL Expression Editor**. Handles are bound to GUI controls using standard .NET databinding technology.

Runtime Functionality

The following table shows the runtime functionality of the ECO framework

Runtime Feature	Notes
Object persistence	Access with the IPersistenceService interface in `Borland.Eco.Services`.
OCL queries and OCL expression evaluation	Access with the IOCLService interface in `Borland.Eco.Services`.
Undo/Redo mechanism	Access with the IUndoService interface in `Borland.Eco.Services`.
Subscription mechanism	Accessed through the IOCLService interface and through classes defined in the `Borland.Eco.Subscription` namespace.
Object version mechanism	The ECO framework has support for saving multiple versions of an object. Access this feature through the IVersionService interface in `Borland.Eco.Services`.
Model introspection	Access information about the model using interfaces defined in the `Borland.Eco.UmlRt` namespace. You can access the type system of the model through the TypeSystem property of the EcoSpace class.

ECO Modeling Tools Overview

This topic describes the integration of the ECO framework with Developer Studio 2006:

- ECO projects and code templates
- Integration with the **Model View**, **Tool Palette**, and **Object Inspector**
- Structural modeling with ECO UML class diagrams
- Behavioral modeling with ECO state machine diagrams

Before reading this topic you should be familiar with ECO framework terminology discussed in the Overview of the ECO framework. Please refer to the link below for more information.

ECO Modeling Tools in Developer Studio 2006

The ECO framework is tightly coupled with the TogetherLiveSource modeling tools. You can model structural features using the **ECO class diagram**, and behavioral features using the **ECO state machine diagram**. The IDE generates ECO-enabled source code as you work with the diagrams.

All of the capabilities of LiveSource are available in ECO projects, such as generation of diagrams from non-ECO source code, navigation from the diagram to source code, layout tools, printing and exporting diagrams to images. These tools all work through coordination between the **Model View**, **Diagram Views**, the **Tool Palette**, and the **Object Inspector**.

ECO Projects and Code Templates

The IDE has code-generating templates to help you develop ECO applications. The following project creation templates are available for Delphi or .NET and C# applications:

Template	Purpose
ECO WinForms Application	Creates an application with a default ECO space, a root ECO UML package, and an ECO enabled Windows form.
ECO ASP.NET Web Application	Creates an ASP.NET application with automatic ECO space pooling.
ECO ASP.NET Web Service Application	Creates an ASP.NET web service with automatic ECO space pooling.
ECO Package in a DLL	Creates a project with a root ECO UML package, but no ECO space.
ECO Package in package (Delphi for .NET)	You can reference the ECO Package DLL in another project, to make the entire model available for use in that application.

The following file creation templates are for use in existing ECO projects:

Template	Purpose
ECO Enabled Windows Form	Adds an ECO enabled Windows form to your project.
ECO Space	Creates a new subclass of DefaultEcoSpace in your project.
ECO PersistenceMapperProvider	Creates a new persistence mapper provider in your project. A persistence mapper provider specifies the persistence mechanism and persistence configuration for the application. You can connect multiple ECO spaces to a single persistence mapper provider.

Code generated by these templates will include all of the necessary ECO-related .NET attributes and default interface implementations.

Working with ECO in the Model View Window

Using the **Model View** window you can navigate your project based on the logical relationships between the classes and other elements in source code. LiveSource scans source code and derives the elements, such as namespaces and classes, and the relationships between them. Because it gives you an unfiltered view by design, LiveSource will expose some implementation details behind the ECO framework.

On a LiveSource class diagram you will see ECO UML packages represented as classes within your project namespace. On an **ECO class diagram**, you will see the true, logical representation of the UML package.

ECO UML packages appear under the root node of the project in the **Model View** (you can also nest packages within other packages). The default name of a new ECO UML package is `Package_N` where `N` is an integer. In the **Model View**, an ECO UML package node (and all ECO UML packages underneath it) is distinguished from a .NET namespace node by its icon. The icon represents an ECO UML package.

You work with ECO and TogetherLiveSource using context menus, which are available on both the **Model View** and on diagrams. The menus are context sensitive, so they automatically reflect only those operations that are valid on the selected element. For example, if you right-click on a class in either the **Model View** or the diagram, the context menu will contain choices for adding attributes and operations. These menu items would not be available if you right-clicked on an ECO UML package; the context menu for a package contains its own set of menu items.

Once you have created a diagram in the **Model View**, you can also add elements to it by dragging them from the **Tool Palette**.

You can set properties of any element by selecting it and editing its properties in the **Object Inspector**. You may select the element either in the **Model View**, or on the diagram. As you work with elements in the **Model View**, diagrams, and the **Object Inspector**, the IDE generates source code to implement the model.

Structural Modeling with the ECO Class Diagram

You draw the structural features of your model using the **ECO class diagram**. The **ECO class diagram** supports these modeling features:

- Creating ECO UML packages
- Creating ECO classes
- Drawing generalization (inheritance) links between classes
- Drawing associations between classes
- Attaching notes to diagram elements
- Adding attributes and operations to classes

Class diagrams are created and opened from the **Model View**. Each ECO UML package has its own primary class diagram with the same name as the package. The primary class diagram cannot be deleted. The primary class diagram for a UML package always shows the entire contents of the package; it displays all of the sub-packages, classes, and relationships that exist within that package. When you add a new class to a UML package it is automatically represented on the primary class diagram.

You can also create secondary class diagrams within a ECO UML package, if you want to show a subset of the classes within the package. New elements added to the package are not automatically added to secondary diagrams. Secondary diagrams can be renamed and deleted.

Any UML elements added to a primary or secondary class diagram will be contained within the UML package that owns the diagram. To show elements in other UML packages, you must create a shortcut to the element. You can do this through the context menu of the class diagram. Shortcuts are displayed on the diagram with a small arrow icon in their lower left corner. Once a shortcut has been created, you can add associations between it and the classes in the UML package that owns the diagram.

Behavioral Modeling with the ECO State Machine Diagram

While the **ECO class diagram** allows you to model structure, the **ECO state machine diagram** allows you to model behavior. The **ECO state machine diagram** supports these features:

- Creating states and state transitions
- Creating entry and exit actions to be performed on entering and leaving a state
- Adding trigger methods to classes
- Creating Effects, which are executed when a state transition occurs
- Creating composite states
- Modeling concurrent states by adding additional regions to a state

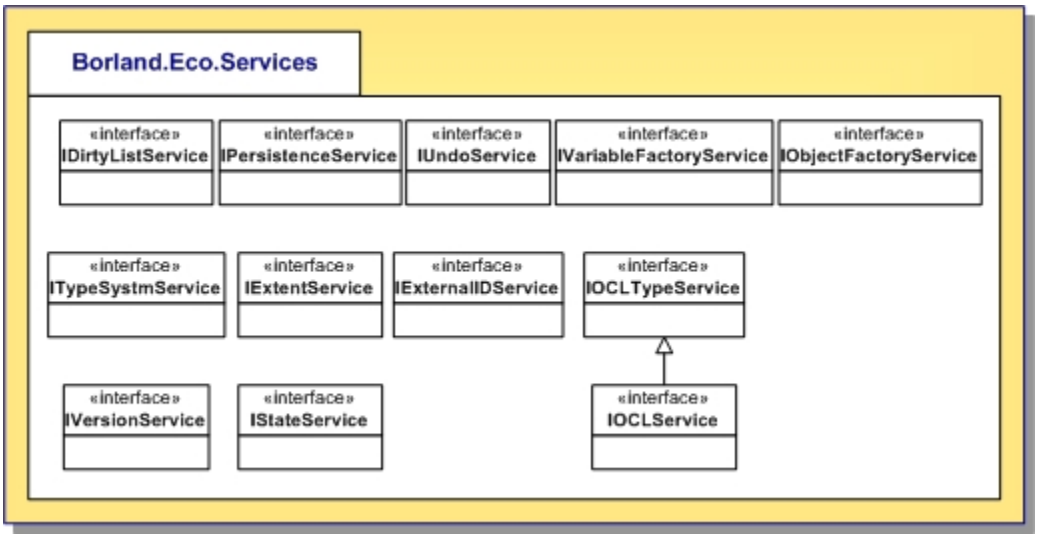
Please see the links below for more information on working with ECO state machine diagrams.

Working with the ECO Service API

This topic describes how to access the ECO framework service API. Code examples demonstrate how the services are exposed through the application's ECO space, as well as how to call methods on an interface. The following concepts are covered:

- Service API Overview
- Accessing the ECO Space
- Accessing the Service API.

The Borland.Eco.Services Namespace



Each ECO service is declared in the `Borland.Eco.Services` namespace. Individual services are listed in the table below.

Service API Overview

All programmatic access to the ECO framework is done through ECO services. ECO services make it easier to find what you need by collating the substantial functionality of the framework into groups of logically related functions, or interfaces. Each service interface is accessible as a property in the ECO space object of the application. When you create a new ECO framework application using one of the Developer Studio 2006 wizards, the IDE defines an ECO space class for you. The generated class contains property accessors that return an instance of the requested interface. You then use that instance to call methods of the interface. The following is an example of an ECO space class generated by the **New ECO Windows Forms Application** wizard. In the code, notice the read-only properties that expose each interface.

```
[Delphi]
TProject10EcoSpace = class(Borland.Eco.Handles.DefaultEcoSpace)
  private
    procedure InitializeComponent;
    class var fTypeSystemProvider: ITypeSystemService;
    class var fTypeSystemProviderLock: Tobject;
  strict protected
    function GetTypeSystemProvider: ITypeSystemService; override;
  public
```

```
    constructor Create;
    class constructor Create;
    class function GetTypeSystemService: ITypeSystemService; static;
    procedure UpdateDatabase;
    function get_PersistenceService: IPersistenceService;
    property PersistenceService: IPersistenceService read get_PersistenceService;
    function get_DirtyListService: IDirtyListService;
    property DirtyListService: IDirtyListService read get_DirtyListService;
    function get_UndoService: IUndoService;
    property UndoService: IUndoService read get_UndoService;
    function get_TypeSystemService: ITypeSystemService;
    property TypeSystemService: ITypeSystemService read get_TypeSystemService;
    function get_OclService: IOclService;
    property OclService: IOclService read get_OclService;
    function get_ObjectFactoryService: IObjectFactoryService;
    property ObjectFactoryService: IObjectFactoryService
             read get_ObjectFactoryService;
    function get_VariableFactoryService: IVariableFactoryService;
    property VariableFactoryService: IVariableFactoryService
             read get_VariableFactoryService;
  end;
```

```
[C#]
public class Project10EcoSpace: Borland.Eco.Handles.DefaultEcoSpace
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        private void InitializeComponent()
        {
        }

        public Project10EcoSpace(): base()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose (bool disposing)
        {
            if (disposing)
            {
                Active = false;
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        private static ITypeSystemService typeSystemProvider;
        public static new ITypeSystemService GetTypeSystemService()
        {
        if (typeSystemProvider == null)
                lock(typeof(Project11EcoSpace))
```

```
                {
                    if (typeSystemProvider == null)
                        typeSystemProvider = MakeTypeService(typeof(Project11EcoSpace));
                }
            return typeSystemProvider;
        }
        protected override ITypeSystemService GetTypeSystemProvider()
        {
            return Project10EcoSpace.GetTypeSystemService();
        }
        //
        // Services
        //
        public IPersistenceService PersistenceService
        {
            get { return (IPersistenceService)GetEcoService(typeof(IPersistenceService)); }
        }
        public IDirtyListService DirtyListService
        {
            get { return (IDirtyListService)GetEcoService(typeof(IDirtyListService)); }
        }
        public IUndoService UndoService
        {
            get { return (IUndoService)GetEcoService(typeof(IUndoService)); }
        }
        public ITypeSystemService TypeSystemService
        {
            get { return (ITypeSystemService)GetEcoService(typeof(ITypeSystemService)); }
        }
        public IOclService OclService
        {
            get { return (IOclService)GetEcoService(typeof(IOclService)); }
        }
        public IObjectFactoryService ObjectFactoryService
        {
            get { return (IObjectFactoryService)GetEcoService(typeof
(IObjectFactoryService)); }
        }
        public IVariableFactoryService VariableFactoryService
        {
            get { return (IVariableFactoryService)GetEcoService(typeof
(IVariableFactoryService)); }
        }
        //
        // Misc helper functions
        //
        public void UpdateDatabase()
        {
            if ((PersistenceService != null) && (DirtyListService != null))
            {
              PersistenceService.UpdateDatabaseWithList(DirtyListService.AllDirtyObjects());
            }
        }
    }
```

Accessing the ECO Space

Every ECO framework application created by a Developer Studio 2006 wizard has a single instance of the generated ECO space class. The ECO space instance is exposed as a property of the main form. Below is an example of the `EcoSpace` property in a generated main form class:

```
[C#]
public Borland.Eco.Handles.EcoSpace EcoSpace
{
            get { return (Borland.Eco.Handles.EcoSpace)rhRoot.EcoSpace; }
            set { rhRoot.EcoSpace = value; }
}
```

```
[Delphi]

property EcoSpace: TProject10EcoSpace read get_EcoSpace;
```

When you add more ECO-enabled forms to your application using the **ECO Enabled Windows Form** wizard, the IDE will generate a new form class with a constructor that takes an instance of an ECO space as a parameter. In addition, and similar to the main form, each subsequent ECO enabled windows form you create with the wizard will have its own `EcoSpace` property. The constructor initializes this property with the ECO space parameter. An ECO application only has one instance of an ECO space, so the typical usage scenario is to pass the ECO space instance from the main form to secondary forms when they are created. The following example creates a new ECO enabled form in response to a button click on the main form:

```
[Delphi]
// TWinForm is the application's main form.
procedure TWinForm.Button1_Click(sender: System.Object; e: System.EventArgs);
var
  // TWinForm1 is a secondary form generated by the ECO Enabled Windows Form wizard.
  newForm: TWinForm1;
begin

  // Create the secondary form, passing the EcoSpace property to the secondary form's
constructor.
  newForm := TWinForm1.Create(EcoSpace);
  // ...
end;
```

```
[C#]

private void button1_Click(object sender, System.EventArgs e)
{
    // EcoWinForm is a secondary form generated by the ECO Enabled Windows Form wizard.
          EcoWinForm newForm;

    // Create the secondary form, passing the EcoSpace property to the secondary form's
constructor.
             newForm = new EcoWinForm(EcoSpace);
             //...
}
```

Accessing the Service API

The following code demonstrates various ways to call service API methods.

```
[C#]
private void button1_Click(object sender, System.EventArgs e)
{
          IUndoService undoService;

    // Get a reference to the ECO Undo Service.
             undoService = EcoSpace.UndoService;

    // Call the interface's StartUndoBlock method.
             undoService.StartUndoBlock("Undo_Block_1");

    // You can also call directly through the ECO space.
    EcoSpace.UndoService.StartTransaction();

    // ...

    undoService.CommitTransaction();

}
```

```
[Delphi]
procedure TWinForm.Button1_Click(sender: System.Object; e: System.EventArgs);
var
  undoService : IUndoService;
begin

   // Get a reference to the ECO Undo Service.
   undoService := EcoSpace.UndoService;

   // Call the interface's StartUndoBlock method.
   undoService.StartUndoBlock('Undo_Block_1');

   // You can also call directly through the ECO space.
   EcoSpace.UndoService.StartTransaction;

   // ..

   undoService.CommitTransaction;
end;
```

Other service interfaces and their methods can be called using a similar technique. Each ECO service interface and its purpose is shown in the following table.

Interface	Description
IStateService	Allows you to discover whether a particular object or property in the ECO space has been modified.
IPersistenceService	Provides a consistent API for you to update objects in the ECO space, without regard to the persistence mechanism.
IDirtyListService	Allows you to retrieve a list of all modified objects, and to query the ECO space to discover whether any objects have been modified. An object is considered modified if it does not have the same state in memory as in persistent storage.
IExtentService	Allows you to query the ECO space for all instances of a certain class.
IObjectFactoryService	Provides methods for you to create new instances of the classes in your model.

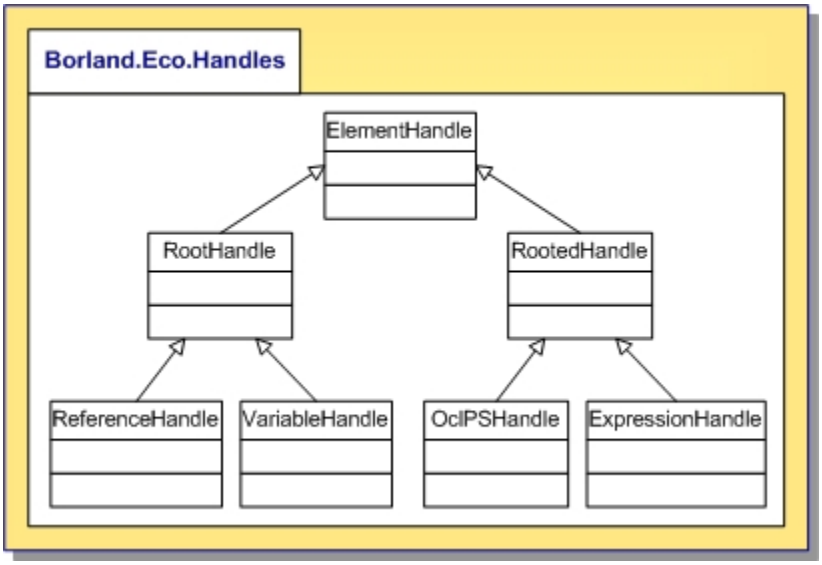
	The IObjectFactoryService interface methods create new objects using their type information. This approach is more generic than directly creating a new object by calling the C# `new` method, or the Delphi `Create` method.
IVariableFactoryService	Provides a programmatic interface for creating what are essentially VariableHandle components. Variables created with this service can be used directly with the IOclService.
ITypeSystemService	Allows you to get the type system of the model, and to validate the model programmatically.
IVersionService	For domain classes that have been marked as versioned, this interface allows you to get a specific version of an object from persistent storage.
IOclService and IOclTypeService	These interfaces allow you to evaluate expressions in Object Query Language (OCL). IOclService is a descendent of IOclTypeService. Only the IOclService interface is exposed through the ECO space.
IUndoService	Allows you to create undo/redo blocks and transactions.
IExternalIdService	Returns a globally unique ID for an ECO object, regardless of whether the object has been saved in persistent storage. This ID is only valid within the ECO space where the ID originated. This service is intended primarily for use in ASP.NET applications.

Working with ECO Handles

This topic introduces the concept of ECO handles, and describes their usage in the ECO framework. Note that further usage of the word *handle* always refers to an ECO handle. Before reading this topic you should have some familiarity with the basics of building Object Constraint Language (OCL) expressions. In particular, the concept of evaluating an OCL expression in a specific context.

- Definition of a root handle and a rooted handle.
- Chained evaluation of handles.
- Usage of handles on the **Tool Palette**.
- Programmatic access to the objects referenced by handles.

Handles in the ECO framework



The diagram shows the relationships between the various kinds of ECO handles.

Handles and Chained Evaluation

Every ECO framework application must have an instance of an ECO space. The ECO space contains both the model definition, and the objects that are created while your application runs. Handles are a mechanism that enables you to get hold of objects in the ECO space at runtime. A handle can represent either a single object, or a list of objects, or a calculated value.

Note: The **ECO Application wizard** automatically declares an ECO space class, and generates code to create of that class at runtime.

Handles are configured at designtime. Setting the properties of a handle at designtime determines the objects the handle will attach to, or the value the handle will hold at runtime.

Handles are linked together to form a chain. The contextual instance of a particular handle is established by the previous handle in the chain. There are two types of handles in the ECO framework:

- **Root handle**: A root handle exists to establish an initial context for all the other handles in the chain.

- **Rooted handle**: Evaluation of a rooted handle begins in the context established by the previous handle. The previous handle can either be a root handle, or another rooted handle.

Handles represent objects and values, therefore, they are also the link between the ECO space and your application's user interface. All ECO handles can be used as .NET data sources for GUI components. The ECO framework uses standard .NET data binding mechanisms. Once you bind a GUI component to a handle, you can work with the component the same way as you would if it were bound to any other kind of data source.

Root Handles

If rooted handles are the individual links in the chain, then a root handle is the spike that is hammered into the ground to anchor the chain. The ground is your application's ECO space.

There are two important designtime properties of root handles that must be set to establish the initial context: the EcoSpaceType property, and the StaticValueTypeName property.

The EcoSpaceType property points to your application's ECO space. The EcoSpaceType property gives the root handle the type system of the model, and a link to the runtime world where objects live.

The StaticValueTypeName property determines the type of object to which the root handle will refer. This property is used by the IDE during designtime to establish a context for the **OCL Expression Editor**. At runtime, the framework will throw an exception if the root handle is ever set to reference an object that does not match the type set in the StaticValueTypeName property.

At runtime, you can set the Element property of a root handle to refer to a specific object in the ECO space. Root handles are the only handles that have a writable Element property. Evaluation of the rooted handles in the chain begins with the object referenced by the root handle.

Rooted Handles

Rooted handles have a property called Expression. The Expression property is an OCL expression that, when evaluated, produces an object, a set of objects, or an atomic element such as a specific attribute or a calculated value. When we talk about evaluating rooted handles within a certain context, we are actually talking about the context for the handle's OCL expression. The context begins at the root handle, and evolves through the chain of rooted handles.

Types of Root Handles

There are two types of root handles you will encounter in the **Enterprise Core Objects** category on the **Tool Palette**. These are the ReferenceHandle, and the VariableHandle classes.

ReferenceHandle

The ReferenceHandle is a concrete descendent of the RootHandle class. The EcoSpaceType property must be configured at designtime to refer to your application's ECO space. The handle's StaticValueTypeName property should also be configured, as this will provide additional designtime assistance in the **OCL Expression Editor**, as well as runtime type checking on the handle's Element property.

Every form that needs access to the objects in the ECO space must have at least one instance of a ReferenceHandle. The **ECO Application wizard** automatically generates a ReferenceHandle for the main form. The default name of this ReferenceHandle is `rhRoot`. For secondary forms, the **ECO Enabled Windows Form wizard** generates a ReferenceHandle, also having the default name `rhRoot`.

VariableHandle

Unlike a ReferenceHandle, a VariableHandle holds a value that does not exist in the ECO space. You configure a VariableHandle with an ECO space and a StaticValueTypeName, however, a VariableHandle is typically not used

to reference objects of classes defined in the model. Instead, a VariableHandle holds values of atomic data types such as the .NET type `System.Int32`. This is because a VariableHandle holds an indirect reference to the object, unlike a ReferenceHandle, which holds the object directly.

A VariableHandle can be used as a data source for GUI components; they are typically used in conjunction with OclVariables objects to create parameters for use in OCL expressions.

Types of Rooted Handles

There are two types of rooted handles you will encounter in the **Enterprise Core Objects** category of the **Tool Palette**. These are the ExpressionHandle, and the OclPSHandle classes.

ExpressionHandle

An ExpressionHandle references an object or a list of objects through the evaluation of its OCL expression.

You must link the RootHandle property of an ExpressionHandle with either a root handle (an instance of a ReferenceHandle or VariableHandle class), or another ExpressionHandle.

You configure the Expression property of the ExpressionHandle using the **OCL Expression Editor**. When you open the **OCL Expression Editor**, the context of the expression (the type of the OCL keyword **self**) is determined by the type of the result returned by the previous handle in the chain. If the previous handle is a root handle, the type is determined from the StaticValueTypeName property. If it is another ExpressionHandle, the type is determined from the Expression property of that handle.

OclPSHandle

Unlike an ExpressionHandle, an OclPSHandle is always executed against persistent storage, rather than data in memory (i.e. in the ECO space). Therefore, the result of executing an OclPSHandle is a static snapshot of the contents of persistent storage.

An OclPSHandle has a method called Execute. The handle's OCL expression is not evaluated until the Execute method is called. Usually, you will call the Execute method in response to some event on a form, such as a button click.

An OclPSHandle is typically used when the OCL expression has an intermediary part that results in a large number of objects, and a subsequent part that filters the set down to a smaller number. For example, a call to `allInstances` followed by a `select` statement.

The OCL expression is first mapped to a SQL query, which is then evaluated by the database. A `select` statement in an OclPSHandle will therefore be able to take advantage of any indices defined within the database. With an ExpressionHandle, the entire set of objects would be created and then processed in memory.

Since the OCL expression of an OclPSHandle is first mapped to SQL, there are some restrictions on OCL constructs that you can use. The following operations and constructs are supported:

- **Navigation:** You can freely access attributes and roles defined in the model. However, derived and non-persistent attributes and roles cannot be used in the expression, since the database has no knowledge of them.
- **List operations:** `select`, `reject`, `allInstances`, `size`, `orderBy`, `minValue`, `maxvalue`, `average`, `sum`, `exists`, `forall`, `notEmpty`, `isEmpty`, and `union` are supported.
- **Boolean operators:** `=`, `<`, `>`, `<=`, `>=`, `<>`, `and`, `or`, `not`, `xor`, `sqlLike`, `sqlLikeCaseInsensitive` are supported.
- **Arithmetic operators:** `+`, `*`, `/`, `-`, `div`, `mod` are supported.
- **Enum:** Enumerated constants are supported.
- **Type operations:** `oclIsKindOf`, `oclIsTypeOf`, `oclAsType` are supported.

- **Other operations:** `IsNull` is supported.

The following operations and constructs are not supported:

- **Typecasting and metadata operations:** `TypeName`, `attributes`, `associationEnds`, `superTypes`, `allSuperTypes`, `allSubClasses`, `oclType` are not supported.
- **String, Date, and numeric conversion:** `subString`, `pad`, `postPad`, `formatNumeric`, `formatDateTime`, `strToDate`, `strToTime`, `strToDateTime` are not supported.
- **Operations relating to Object Versioning Extension:** `atTime`, `allInstancesAtTime`, `existing` are not supported.
- **List operations:** `count`, `includesAll`, `difference`, `including`, `excluding`, `symmetricDifference`, `asSequence`, `asBag`, `asSet`, `append`, `prepend`, `subSequence`, `at`, `first`, `last`, `orderDescending`, `sumTime` are not supported.
- **Other operations:** `length`, `min`, `max`, `asString`, `allLoadedObjects`, `regExpMatch`, `inDateRange`, `inTimeRange`, `constraints`, `collect`, `if`, `concat` are not supported.

There are other restrictions on the OCL expressions used in a OclPSHandle:

- **Data types:** At no point in the expression can there be a collection of attributes (e.g. `Collection(String)`).
- **TableMapping:** Child mapped tables would complicate the questions generated by the translator since each query must be posed to a number of tables. Currently, it is not possible to refer to attributes/roles that are stored in child mapped tables.
- **Bags:** In the OCL specifications, the expression `Person.allInstances.home` should result in a bag of objects. Bags allow for multiple instances of the same object, so if two persons live in the same house, the house would occur twice in the result. SQL, however, does not allow this when making joins, so the results of such an implicit collect will be a set, and not a bag.

Using the Objects Referenced by Handles

Handles reference objects in the ECO space. A handle could therefore refer to a single object, a list of objects, or it might hold calculated values. Regardless, every handle has a property called Element that you use to get the value of the handle. Since the ECO framework has no knowledge of the types defined in your model, there are commonly used code idioms that allow you to get from the ECO type (held by the handle) to a type defined in your model.

The handle's Element property gives back a reference to the ECO IElement interface. The method AsObject returns the element as a .NET `System.Object`. From there, you can cast the object to a type defined in your model, as shown in the following code. In the code, the variable `rhPerson` is a ReferenceHandle that has been set to refer to an instance of a model class called `Person`.

```
[Delphi]

var
  E : Borland.Eco.ObjectRepresentation.IElement;
  O : System.Object;
  P : Person;
begin
   E := rhPerson.Element;
   O := E.AsObject;
   P := O as Person;
   P.DoSomething;     // Now you can call methods and access attributes of the Person class.

   // This code could be abbreviated...
```

```
   P := (rhPerson.Element.AsObject) as Person;
   P.DoSomething;

   // Abbreviating even more...
   (rhPerson.Element.AsObject as Person).DoSomething;
end;
```

```
[C#]

Borland.Eco.ObjectRepresentation.IElement E;
System.Object O;
Person        P;

   E = rhPerson.Element;
   O = E.AsObject();
   P = O as Person;
   P.DoSomething();  // Now you can call methods and access attributes of the Person class.

   // This code could be abbreviated...
   P = (rhPerson.Element.AsObject) as Person;
   P.DoSomething();

   // Abbreviating even more...
   (rhPerson.Element.AsObject as Person).DoSomething();
```

When the element referenced by the handle is a collection, you must first cast the Element property to the ECO interface IObjectList. In the following code, the variable `ehAllPersons` is an ExpressionHandle. It is also assumed the list returned by this expression contains at least three elements. The Expression property has been set to retrieve all instances of the `Person` class from the ECO space.

```
[Delphi]

var
  L : Borland.Eco.ObjectRepresentation.IObjectList;
  O : System.Object;
  P : Person;

begin
  L := ehAllPersons.Element as IObjectList;  // Cast the element to an IObjectList
  O := L[2].AsObject;                        // Retrieve the object at list index 2, and
cast it to a System.Object
  P := O as Person;                          // Cast the object to a Person
  P.DoSomething;                            // Access properties and methods of the Person
class.

  // This could be abbreviated...
  L := ehAllPersons.Element as IObjectList;
  P := (L[2].AsObject) as Person;
  P.DoSomething;

  // Abbreviating even more...
  P := (ehAllPersons.Element as IObjectList)[2].AsObject as Person;
  P.DoSomething;
end;
```

```
[C#]

Borland.Eco.ObjectRepresentation.IObjectList L;
```

```
System.Object O;
Person P;

  L = ehAllPersons.Element as IObjectList;  // Cast the element to an IObjectList
  O = L[2].AsObject;                       // Retrieve the object at list index 2, and cast
it to a System.Object
  P = O as Person;                          // Cast the object to a Person
  P.DoSomething();                        // Access properties and methods of the Person class.

  // This could be abbreviated...
  L = ehAllPersons.Element as IObjectList;
  P = (L[2].AsObject) as Person;
  P.DoSomething();

  // Abbreviating even more...
  P = (ehAllPersons.Element as IObjectList)[2].AsObject as Person;
  P.DoSomething();
```

Working with ECO Subscriptions

This topic describes how the ECO subscription mechanism is implemented, and how you work with it in your applications. The following items are discussed:

- The ECO subscription mechanism.
- Two different types of subscriptions: Reevaluate and Resubscribe
- Using subscriptions with derived attributes.
- Using the SubscriberAdapterBase abstract class.

The ECO Subscription Mechanism

The ECO framework implements a publish and subscribe pattern to notify subscribers of changes to objects, relations, and attributes.

Note: Objects, relations, and attributes are all implementers of the IElement interface.

The ECO handles that use OCL expressions, such as ExpressionHandle, are programmed to work with the subscription mechanism. When you work entirely within the form designer, using the **OCL Expression Editor** to configure handles on a form, you do not need to be aware of the inner workings of the subscription mechanism at all.

However, there are times when you will want to use the IOclService interface directly. For example, if you have a component that is not aware of the .NET databinding mechanism (such as a status bar) and you want to display values in this component, you will call the EvaluateAndSubscribe method of the IOclService interface. Another example might be to display a special icon when changes have occurred, such as an email program might indicate when unread messages have arrived. Again, you would use the EvaluateAndSubscribe method to accomplish this. Finally, you might also encounter a case where the value of an attribute or column cannot be computed in OCL.

When using the IOclService directly, you must know the two different kinds of subscriptions to which you can respond. When you need to compute a value in source code rather than in OCL, you must know how to place the two different kinds of subscriptions.

Reevaluate and Resubscribe

Looking at the four overloaded IOCLService methods, EvaluateAndSubscribe, you can see that each one takes two different subscriber parameters: `reevaluateSubscriber`, and `resubscribeSubscriber`.

```
IElement EvaluateAndSubscribe(IElement root, string expression, ISubscriber
reevaluateSubscriber, ISubscriber resubscribeSubscriber);
IElement EvaluateAndSubscribe(IElement root, IExternalVariableList variableList, string
expression, ISubscriber reevaluateSubscriber, ISubscriber resubscribeSubscriber);
IElement EvaluateAndSubscribe(IElement root, IClassifier rootType, string expression,
ISubscriber reevaluateSubscriber, ISubscriber resubscribeSubscriber);
IElement EvaluateAndSubscribe(IElement root, IClassifier rootType, IExternalVariableList
variableList, string expression, ISubscriber reevaluateSubscriber, ISubscriber
resubscribeSubscriber);
```

These two parameters correspond to the two different kinds of subscriptions you can place: *Reevaluate* subscriptions, and *ReSubscribe* subscriptions. The difference between them has to do with the impact any change in the ECO space has on existing subscriptions. All changes will always cause a reevaluation to occur, so that subscribers will be informed when they must reevaluate a particular data value. In addition to the reevaluation of data, some changes in the ECO space also require additional subscriptions to be created. The difference between the two kinds of subscriptions is illustrated in the following example.

You have a model that contains a `Person` class and a `Building` class. You have drawn an association between these two classes such that a person can own zero or many buildings. In addition, you have an association between a `Building` and a `Person`, such that a building can have zero or many residents (i.e. instances of the `Person` class). These relationships are shown below.

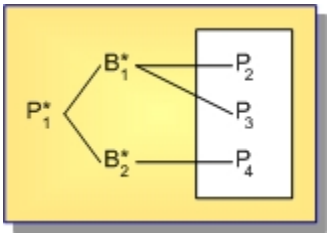
At some point while your application is running, the ECO space contains one person object, and this person owns two buildings. You have built the following OCL expression to retrieve all the residents in all the buildings owned by a person:

```
self.ownedBuildings.residents
```

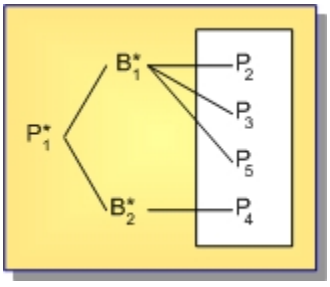
Note: In the expression, `self` is an object of type `Person`.

The purpose of the subscription mechanism is to allow you to keep all the components that display or use data returned by this expression up to date.

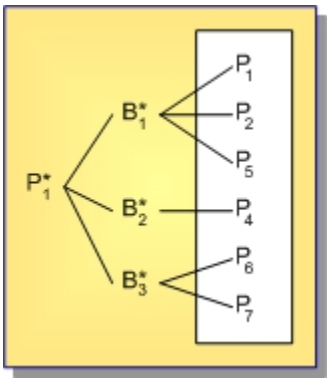
The result of this OCL expression is shown in the diagram. The subscriptions automatically placed by the OCL evaluator are marked with an asterisk.



If a new `Person` is created and added to the list of residents for building B1, the result would be as shown:



Adding a new person as a resident in an existing building changed the result set of our OCL expression, but it did not impact the set of subscriptions itself. This kind of change would trigger only a reevaluate subscription. Adding a new building with its own set of residents would result in the structure shown in the diagram below.



The change not only affected the result set, but it caused a new subscription to be added as well. This kind of change triggers both a reevaluate and a resubscribe subscription.

The rule of thumb is that if a change occurs in the last element of an OCL navigation (in this example, in the residents relation) only the value needs to be reevaluated (a reevaluation is required). If a change occurs anywhere else in the navigation (in this example, in the `ownedBuildings` relation), both the value and the subscriptions must be reevaluated (a reevaluation and a resubscription are required).

Having two different subscribers allows you to take different actions when these two types of subscriptions occur. When working with the EvaluateAndSubscribe method, you can pass a null value for either subscriber parameter if you are not interested in that kind of subscription. You can also pass the same subscriber to both parameters, causing a minor impact in performance. A resubscription will be performed in those cases where only a reevaluation is required.

Using Subscriptions with Derived Attributes

In some cases you will not be able to compute the value of a derived attribute in OCL. In these cases you must implement a specific design pattern in your class so that the framework will be able to call your source code to get the value of the attribute. For a derived attribute whose value is computed in source code, you must add a method to your class with the following signature:

```
[Delphi]

function attributeNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
```

```
[C#]

System.Object attributeNameDeriveAndSubscribe(ISubscriber reevaluateSubscriber, ISubscriber
resubscribeSubscriber);
```

You must replace `attributeName` with the name of the attribute you are deriving. For example, in our `Person` class, if we wanted to derive the attribute called `fullName` in source code, we would implement the method

```
[Delphi]

function fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
```

```
[C#]
```

```
System.Object fullNameDeriveAndSubscribe(ISubscriber reevaluateSubscriber, ISubscriber
resubscribeSubscriber);
```

Please refer to the procedure *Deriving an Attribute in Source Code* for an example of computing a value and placing subscriptions in source.

Using the SubscriberAdapterBase Abstract Class

If you need to implement a component that responds to subscriptions, start by deriving a subclass of the ECO abstract class, SubscriberAdapterBase. When you use SubscriberAdapterBase, implement its abstract DoReceive method to respond to the subscription.

Typically you will create a private utility class to implement a subclass of SubscriberAdapterBase.

Please refer to the procedure *Implementing a Subclass of SubscriberAdapterBase* for an example.

Using State Machines with the ECO framework

These topics describe how to work with ECO state machine diagrams.

In This Section

Modeling Behavior with State Machines
Describes simple UML state machines and how they are used with the ECO framework.

Using Substates with the ECO framework
Describes how to use composite states, substates and regions with ECO state machine diagrams.

Modeling Behavior with State Machines

This topic introduces the basic terminology of UML state machines, and how state machines are used to model behavior in ECO framework applications.

The following concepts are discussed:

- Definition of a state machine and its associated terminology
- Attributes of a state and of state transitions
- Performing activities in a state machine

Definition and Properties of a State Machine

During its lifetime, an object enters into many different conditions based on the values of its member variables. Each individual condition is called a state. A state transition occurs when an object changes from one state to another. State transitions occur in response to a trigger. In an ECO state machine diagram, the trigger is always a method on the class. Trigger methods are designated by setting the `Is Trigger` property of the method to `True` in the **Object Inspector** (the method may be selected in the **Model View**, or on the **ECO class diagram**).

The actions associated with a state transition typically set member variables and call other methods on the object, thus taking the object from one state to another.

A state machine diagram shows the states and transitions of the object, together with their associated attributes such as entry and exit actions, trigger events, and guard conditions. Modeling behavior with ECO framework state machine diagrams allows you to concentrate on the business logic that drives your application, rather than on the internal mechanics of setting member variables. The framework handles the mechanics; you handle the logic.

Properties of a State

States and state transitions have properties that you set using the **Object Inspector**. These properties describe the behavior of the object from its creation to deletion. A state has these properties:

Name	A text string that describes the state.
Entry and Exit actions	Executable actions that are performed upon entry and exit from the state, respectively. Entry and exit actions are written in ECO Action Language.

Properties of a State Transition

A state transition has these properties:

Effect	An activity that occurs when the transition takes place. The effect of a state transition is written in ECO Action Language.
Guard	A Boolean expression that indicates whether the transition can occur or not. The guard written in the Object Constraint Language (OCL). A guard with no OCL expression is called an empty guard, and is always `True`.
Trigger	A method that causes the state transition to occur. The trigger must be an operation on the class holding the state machine. A trigger just an operation with its `Is Trigger` attribute set to `True`. Since they are a structural feature of classes, triggers are added either on the **Class Diagram**, or using the **Model View**. Trigger parameters can be specified using the **Object Inspector**. The parameters of trigger methods can be used in the guard expression of the transition.

Source (read only)	The source state for the transition. When the transition occurs, the exit action of the client is executed.
Target (read only)	The destination state for the transition. When the transition occurs, the entry action of the supplier state is executed.

Initial and Final States

All UML state machines must have one initial state. The initial state is a pseudostate, since the object is never actually in the initial state. The initial state serves as the source for the outgoing transition that points to the true starting state of the state machine. The initial state may have multiple outgoing transitions. These transitions may not have triggers, but they may each have a guard. Exactly one of the guards must evaluate to `True`, so the state machine will start in a defined state.

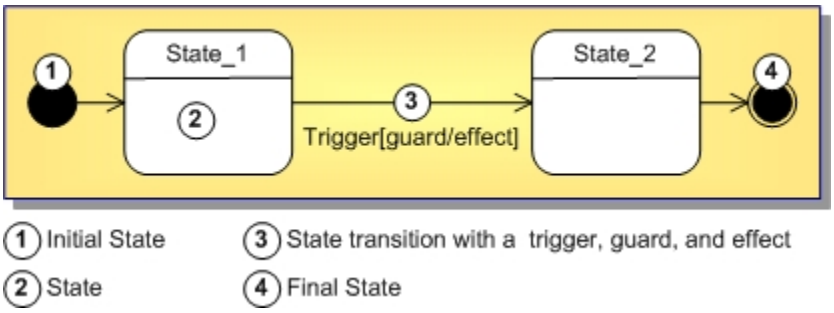
The final state is only of interest when a state machine is “finished” with its task. On an ECO state diagram, the final state implies deletion of the object. In many cases ECO objects never enter a final state, so the final state is not required on an ECO state diagram. The final state has an entry action attribute that you can use to perform final operations. On transition to the final state, the order of execution is

Note: Object deletion occurs in the final state of the state machine as a whole, not in the final state of a region. See the link below on using substates for more information.

1. Effect of the incoming transition to the final state
2. State change to the final state
3. Entry action of the final state
4. Object deletion

A Simple State Machine Diagram

The following simple state machine diagram illustrates the concepts discussed so far.



Activities in ECO framework State Machines

When a state transition occurs, the ECO framework performs activities associated with the source state, the transition itself, and the target state. These activities are defined on the **ECO state machine diagram**, using ECO Action Language.ECO Action Language is an extension of OCL that allows side-effects. Refer to the links below for more information on ECO Action Language.

When the trigger of a state transition trigger is executed and the guard is true, the following activities will happen in order:

1. The exit activity of the source state of the transition is executed.
2. The effect activity of the transition is executed.

3 The ECO framework sets the state of the object.

4 The entry activity of the target state of the transition is executed.

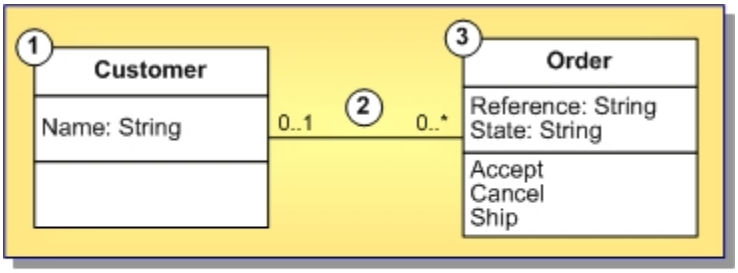
There are two exceptional cases that could occur when a trigger is called, as shown in the following table.

Case	Action
More than one guard expression is true	The framework throws an exception of type AmbiguousTransitionException.
All guard expressions are false	The framework throws an exception of type NoTransitionException.

A Basic ECO State Machine

The following example of a simplified order entry system will tie together the basic concepts of ECO class diagrams and ECO state machine diagrams.

The sample starts with two classes: Customer and Order. The figure shows an ECO class diagram with these two classes. The Customer class contains a string for the customer name, and the Order class contains a reference number. A Customer may have zero or many orders, as shown in the association.



1 Customer class with Name attribute

2 A customer may have zero or may orders

3 Order class with Reference attribute and three trigger methods

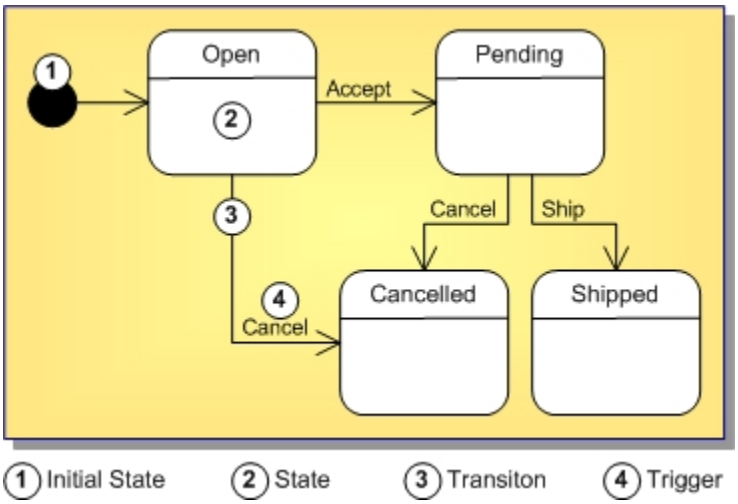
The diagram shows three trigger methods on the Order class: Accept, Cancel and Ship. Trigger methods are operations on the class that have their `Is Trigger` attribute set to true. Like all operations, triggers can take parameters, which you specify on the **Object Inspector**. Trigger parameters may be used in ECO Action Language activities. In the Order system, the Ship trigger takes one parameter, which is the waybill number.

The `State` attribute of the Order class is created automatically when you create a state machine diagram for the class. The attribute is an ordinary attribute of the class, except its `Is state` attribute property is set to `True`.

Tracking the State of an Order

An ECO state machine diagram will be used to model the behavior of an Order object within the system. As shown in the diagram, an Order has the following states

- Open
- Pending
- Shipped
- Cancelled

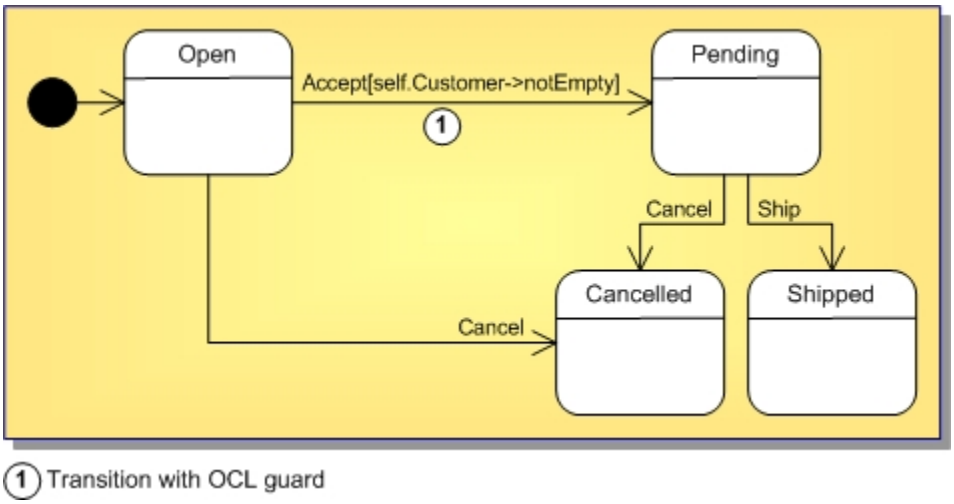


As shown in the diagram, each transition has an associated trigger method. A state transition occurs when you call a trigger method on the object. For example, from the Open state, you may call the Accept trigger, or the Cancel trigger to effect a transition to the Pending or Cancelled state, respectively. Once an Order has entered either the Cancelled or Shipped state, there are no more transitions available. Therefore, the Cancelled and Shipped states are effectively the final states of an Order.

Adding OCL Guards to the State Machine

A guard is a condition on a state transition. Guards are expressed in OCL, and must evaluate to a boolean value. Like all state and state transition attributes, the guard expression is a property that you edit using the **Object Inspector**. A guard with no expression always evaluates to true, meaning the transition will always occur when its associated trigger method is called.

The next diagram shows a guard condition on the Accept trigger. The guard checks to see if there is a Customer associated with the Order. If the Customer attribute of the Order is not a null value, the guard evaluates to `True`, and the trigger is executed thus causing a transition to the Pending state.



Adding Activities to the State Machine

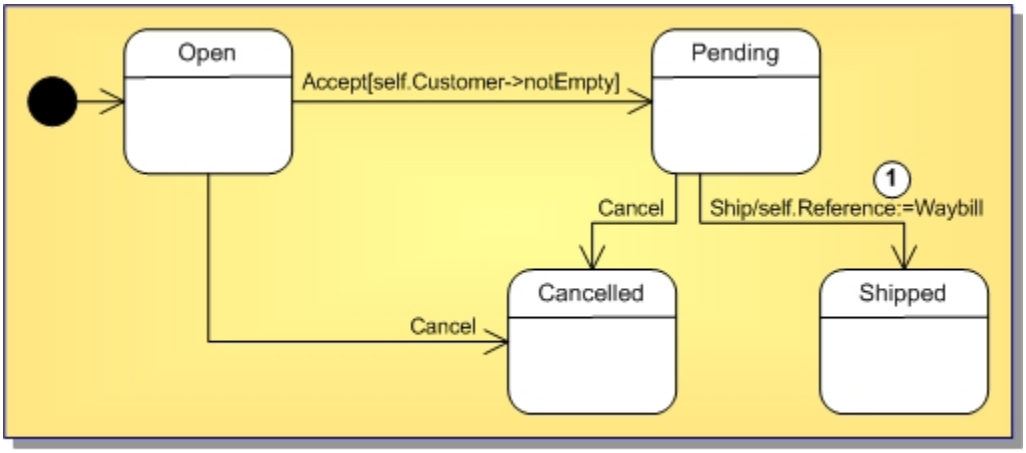
In addition to moving from one state to another, you can associate activities to be performed when:

- A transition trigger is taken
- A state is entered

- A state is exited

Activities are executable actions that are written using ECO Action Language. ECO Action Language is an extension of OCL that allows for side effects. To specify an activity for a transition, you add code to the effect attribute of the transition. To specify activities for states, you add code to the entry and exit attributes of the state.

Every Order in the tracking system requires a waybill number, but this piece of information is not known until the order is shipped. This requirement can be built directly into the model by using the Effect attribute of the Ship transition. The next diagram shows the complete state machine for the Order system, with the Effect on the Ship transition. Note the Effect is written in ECO Action Language, with the `:=` operator assigning the value of the Waybill parameter to the Reference attribute of the Order.



(1) Transition with effect, written in ECO Action Language

All of the procedures necessary to create this simple ECO state machine are referenced at the end of this topic.

Using Substates with the ECO framework

A state with no substates is called a simple state. A composite state is a state that contains nested states, or substates. A composite state can contain either substates (sometimes called sequential, or disjoint substates), or concurrent (orthogonal) substates. Using substates can greatly simplify complex state machine diagrams.

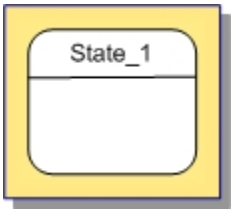
This topic discusses the following:

- Regions and substates
- Composite states and substates
- Concurrent substates

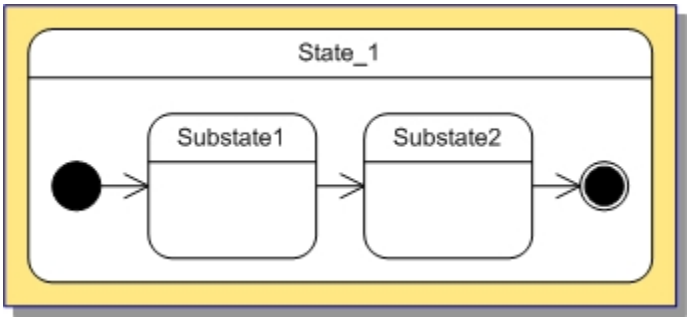
Regions and Substates

Substates exist within regions of their composite state. The substates in a region may also have regions, which may have their own substates. Multiple regions within a composite state are delineated by a dotted line.

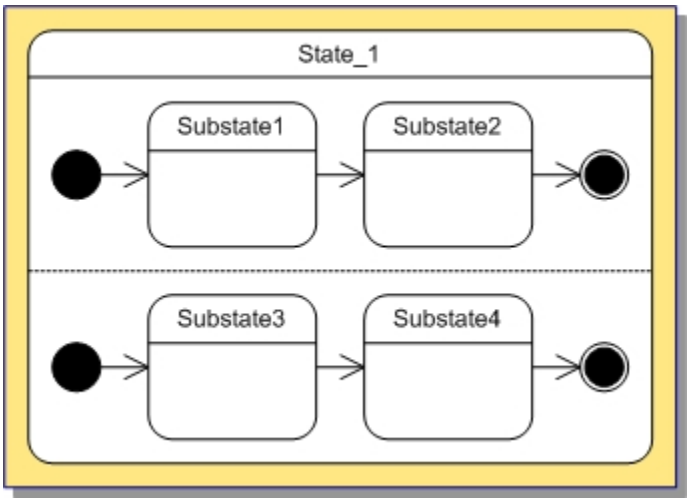
The following diagram shows a simple state with one region.



The next diagram shows a composite state with one region containing substates.



The next diagram shows a composite state with two regions.



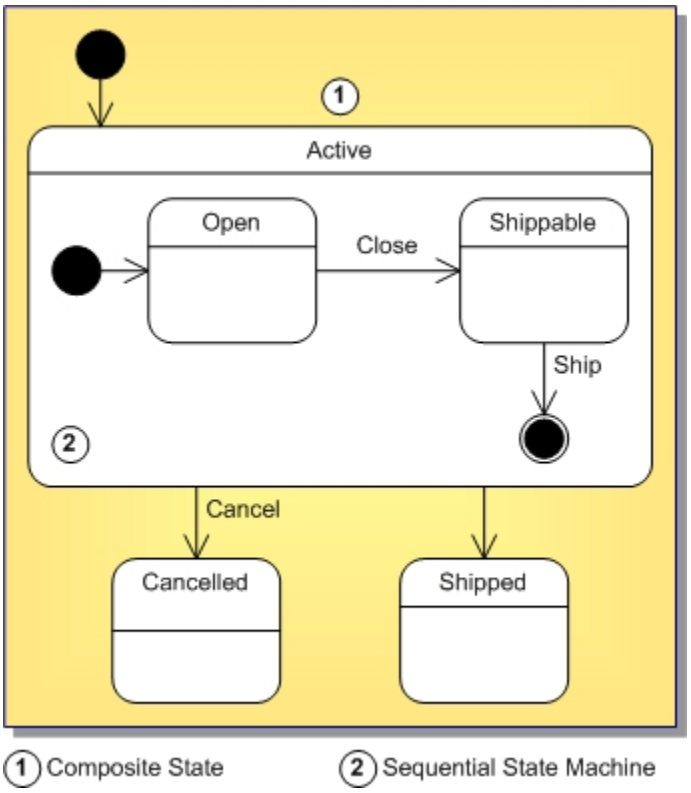
The relationship between regions and concurrent substates is explained below.

Composite States and Substates

Substates are a specialization of the composite state that contains them. Since the object is never actually in the composite state, it is analogous to an abstract class. All of the substates within the composite state inherit the transitions of the composite state. Only one substate can be active when the composite state is active. This is why substates are sometimes called sequential, or disjoint. The composite state represents the condition where the object is in any one of the substates contained within the composite state.

Substates can simplify the state machine when there are multiple transitions leading to the same state. Often, these multiple transitions will each have the same trigger, guard, and effect. For example, this can happen where there are multiple states and a cancel state. With simple state machines, you would have transitions leading out of each state, to the cancel state.

The following diagram shows a state machine drawn on an Order class from a hypothetical order entry system.



The diagram shows a composite state called Active. The Active state contains one region with two substates called Open, and Shippable. When an Order object enters the Active state, it immediately enters the initial state of the nested state machine. In this case, the Order will enter the Open state.

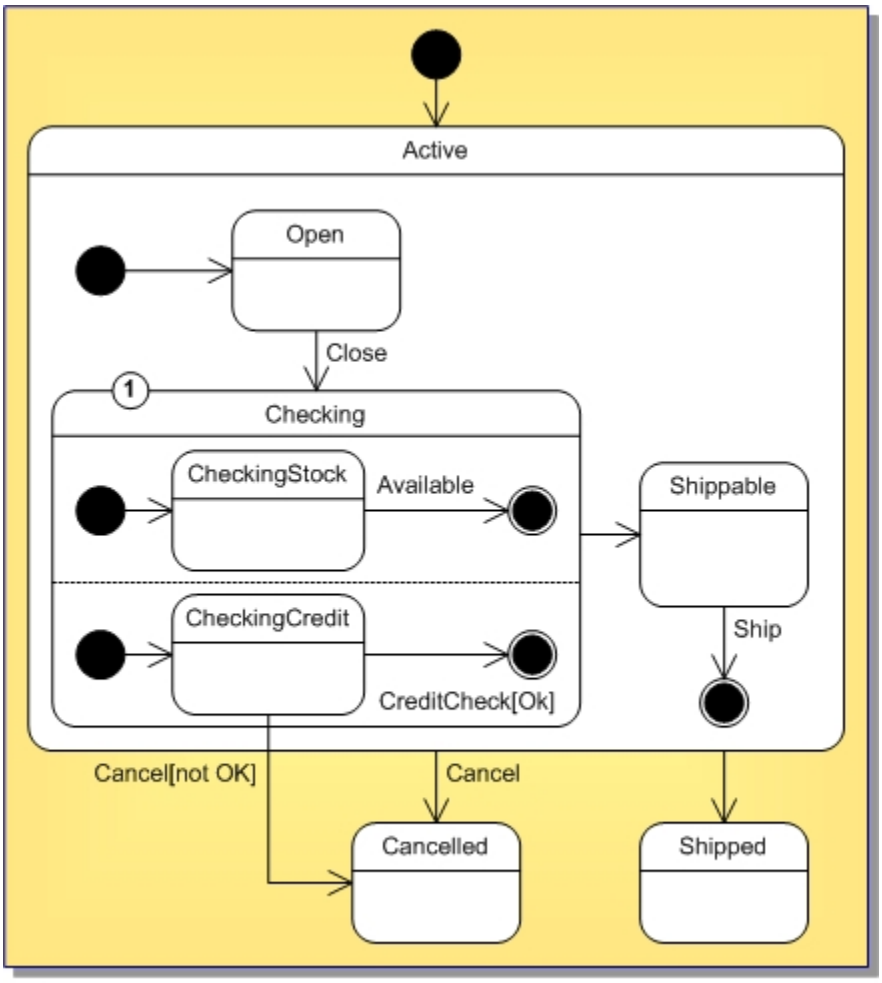
There are two transitions out of the Active state. States inherit the transitions of their composite state, so given this state machine, the transition to the Cancelled state can be reached by calling the Cancel trigger from either the Open, or the Shippable state.

Concurrent Substates

Adding regions to a composite state allows you to create concurrent substates. The state machines in separate regions are concurrent because they execute in parallel. The object is therefore in multiple states simultaneously. On entering the composite state, the object enters the initial state of each of the state machines in the regions it contains.

In the order tracking system, suppose an Order must pass certain checks before it is shipped. For example, suppose the item referenced on the Order must be in stock, and the customer must pass a credit check. You could model these checks with sequential substates, but concurrent substates help simplify the model.

The following diagram shows the enhanced Order state machine.



1 Composite State with two regions

The following list describes notable points about the enhanced state machine diagram:

- To model the two-phase order checking process, a composite state called Checking is introduced. The two checks performed on an Order are represented in two regions within the Checking state. On calling the Close trigger from the Open state, the Order object simultaneously enters the CheckingStock and CheckingCredit substates.
- A single, triggerless transaction leads out of the Checking composite state. When the final state is reached in all regions, the triggerless transaction will be taken. In this case, the Order enters the Shippable state.
- An OCL guard expression called `Ok` is used on the transitions Cancel and CreditCheck. These two transitions will not be taken unless the guard evaluates to `true`.

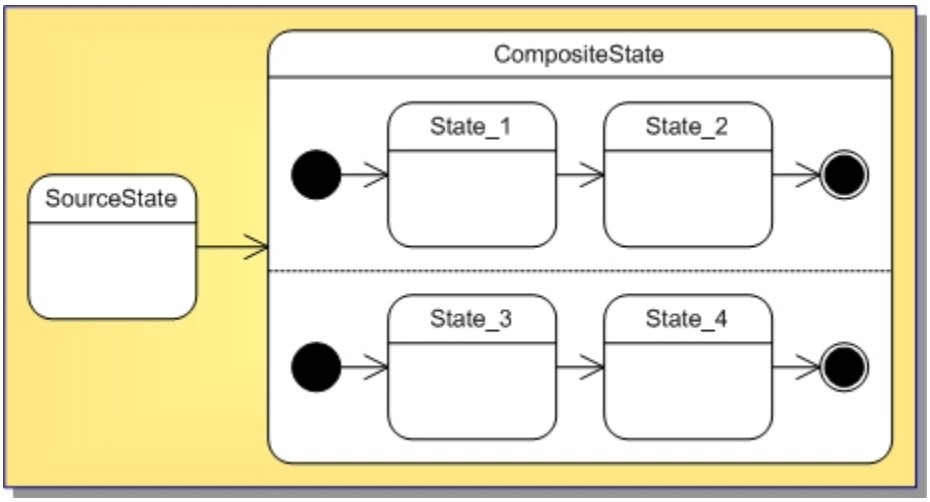
Entry and Exit from Composite States

Composite states can be entered and exited in a variety of ways. Each case is unique in terms of whether an initial or final state is required, and in the way entry and exit actions are called.

Entry into a Composite State

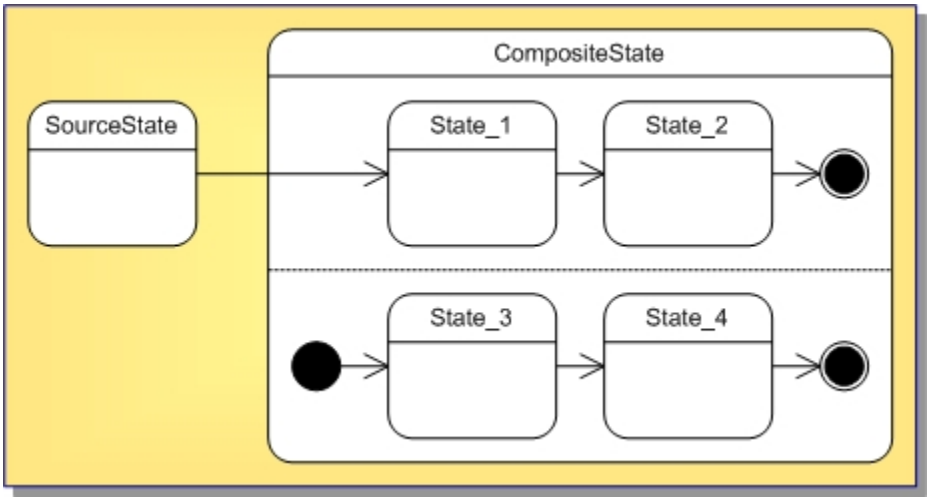
There are two cases of entry into a composite state.

The first case occurs when the target of the transition is the composite state itself, as illustrated in the following diagram:



In this case the state machines in the regions of the composite state must all have an initial state. On entry to the composite state, the object will be in the initial state of each region it contains.

The second case occurs when the target of a transition is a substate in a region of the composite state. This case is illustrated in the following diagram:

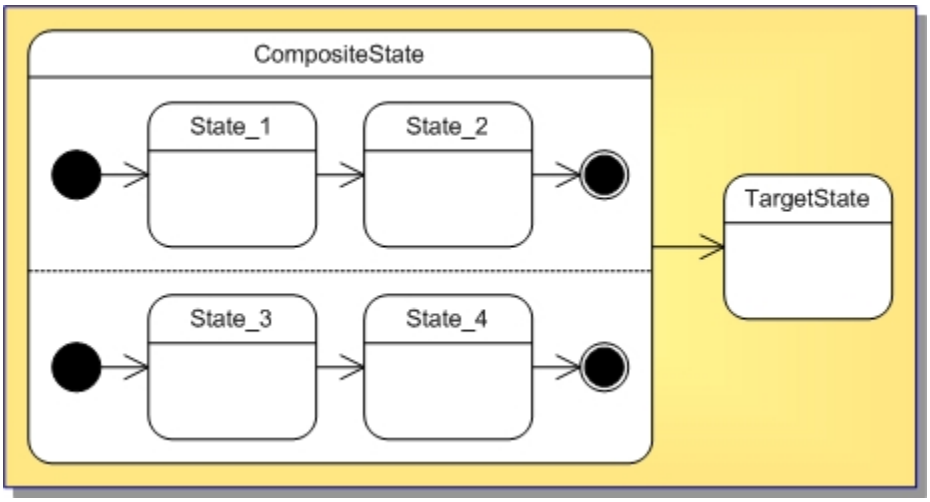


In this case the state machine of the target substate does not require an initial state. On entry to the composite state, the object will be in the target substate, and in the initial state of the other regions it contains.

Exit from a Composite State

There are three cases of exit from a composite state.

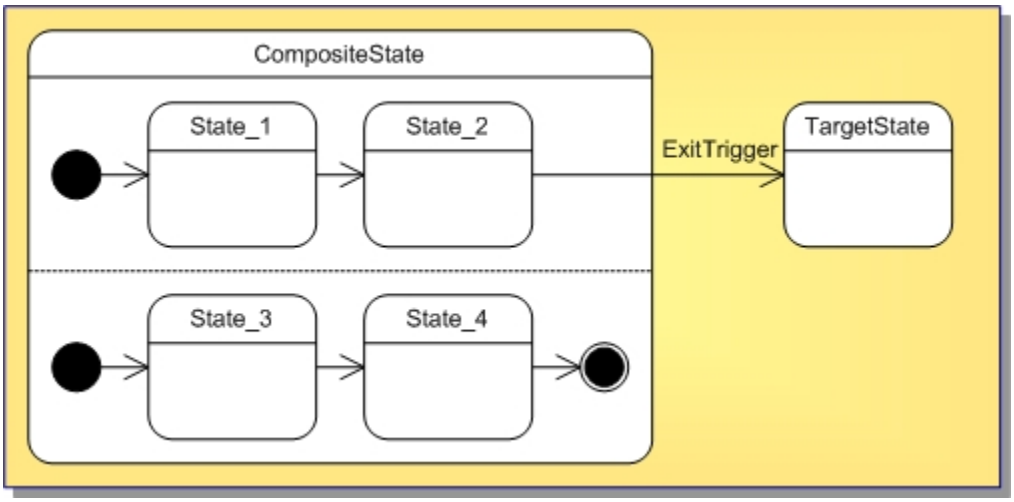
The first case occurs when a triggerless transition leads from the composite state itself, as illustrated in the following diagram:



In this case each of the state machines in regions of the composite state must have a final state. The triggerless transition will occur when the final state of each region is reached.

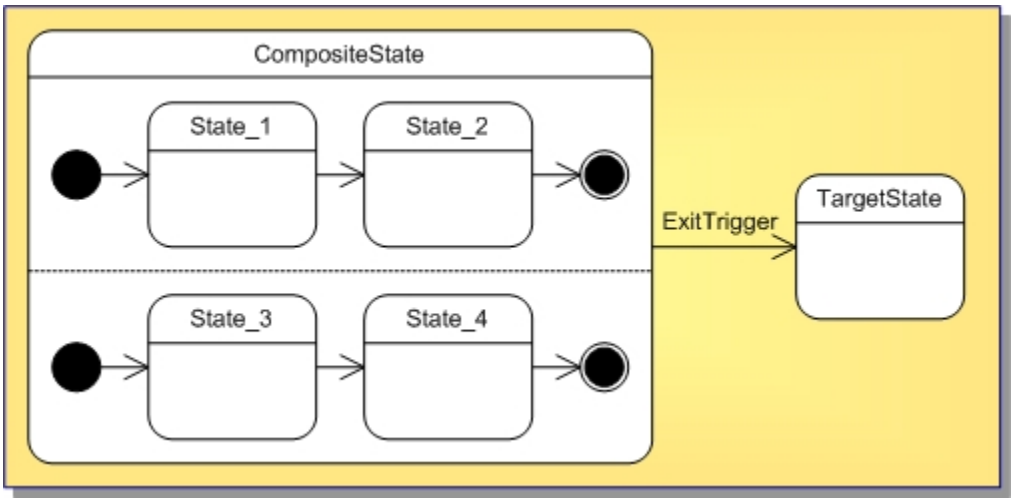
Note: You could have multiple triggerless transitions with OCL guard expressions. Exactly one of the guard expressions must evaluate to `true`, otherwise an AmbiguousTransitionException is thrown.

The second case occurs when there is a transition leading from a substate to a state outside the composite state. This case is illustrated in the following diagram:



As shown in the diagram, the transition must have a trigger. When the trigger is called, the composite state is exited. The exit action of the currently active substate in each region is called, and then the exit action of the composite state is called.

The third case occurs when a transition with a trigger leads from the composite state. This case is illustrated in the following diagram:



When the trigger is called, the exit action of the currently active substate in each region is called, and then the exit action of the composite state is called. The concurrent state machines of the composite state need not have reached their final state when the trigger is called.

Object Constraint Language (OCL) and ECO Action Language

These topics describe the use of Object Constraint Language (OCL) and ECO Action Language with the ECO framework.

In This Section

Overview of the Object Constraint Language
Describes the Object Constraint Language (OCL) in a high-level overview.

Using ECO Action Language
Describes the extensions provided by the ECO Action Language.

Overview of the Object Constraint Language

The Object Constraint Language (OCL) is a formal language used to express constraints in an unambiguous way. Expressions written in OCL can return single values and collections of objects, but they do not alter the state of the objects.

With the ECO framework, you can use OCL throughout the process of designing your application. OCL is available to:

- Specify the values of derived attributes of classes.
- Specify the values of derived association ends.
- Specify constraints for your objects.
- Configure ECO handles, specifying the values or objects to which the handles refer.
- Evaluate dynamically built OCL queries and expressions at runtime.
- Translate OCL queries to SQL to be efficiently executed in the database.

This topic presents a brief introduction to OCL. The official OCL specification in PDF form can be found at the Object Management Group (OMG) website.

Establishing the Context of an OCL Expression

OCL expressions are evaluated in the context of a data type. Typically this type is a class declared in the model. Within the expression, the keyword self refers to the instance of that data type. For example, given the expression:

```
self.Name
```

If the context is a class called `Person`, the self refers to an instance of the `Person` class. This OCL expression returns the value of the `Name` attribute of the `Person` instance.

Knowing the context of the expression is particularly important when configuring ECO handles. Refer to the links below for more information on working with ECO handles.

Note: OCL is a case-sensitive language. Language keywords are lowercase (or camel caps, starting with a lowercase letter). Model elements must be capitalized as they appear in the model.

Predefined OCL Types

The OCL Specification includes four predefined, basic data types. These are:

- Integer
- Real
- Boolean
- String

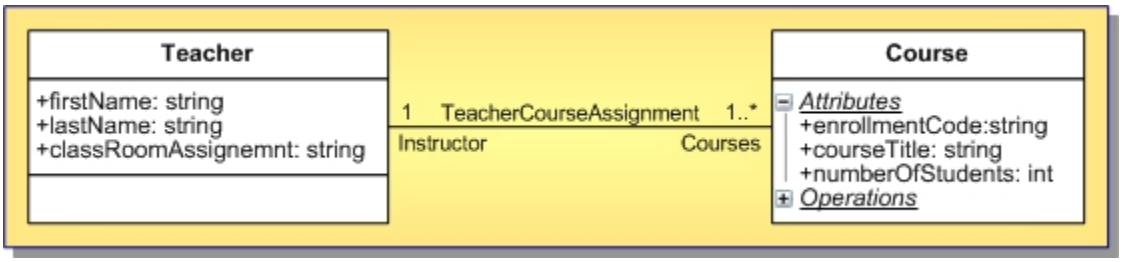
In addition to these intrinsic types, OCL expressions can refer to any data type defined in the model. These additional types can be either values or objects.

The OCL specification includes basic collection types.

Set	Unordered, no duplicates
Sequence	Ordered, no duplicates
Bag	Unordered, duplicates allowed

The Basic Anatomy of an OCL Expression

An OCL expression consists of a context, and navigation from that context to the target value you are interested in. For example, consider the following class diagram:



If the context is `Teacher`, the following are valid OCL expressions.

Expression	Result
`self.firstName`	Value of type `string`
`self.lastName`	Value of type `string`
`self.classRoomAssignment`	Value of type `string`
`self.Courses`	Collection of `Course` objects (because the multiplicity of the association is 1..*)
`self.hireDate`	Results in a `DateTime` object
`self.Courses.numberOfStudents`	Collection of integer values (one for each course)
`self.Courses.numberOfStudents->sum`	Value of type integer (the total number of students in the courses taught by the teacher)

Given the expression:

```
self.Instructor
```

If the context is `Course`, the expression results in a single instance of a `Teacher` (because the multiplicity of that association is 1). If the multiplicity had been 0..1, the result would have been either an instance of `Teacher`, or a null value.

The ECO framework handles navigation on NULL values differently than you might expect. Given the expression:

```
self.Instructor.firstName
```

In the context of a `Course`, this expression is valid and returns the name of the instructor if there is one, or an empty string if the `Course` has no instructor.

Expressions are evaluated from left to right to get the type and value of the result.

OCL with Association Classes

You use the dot operator to navigate to an association class, the same way as navigating to an attribute or association end. The difference is that you follow the dot with the name of the association class, starting with a lowercase letter.

Suppose the association above had an association class of type `Room`. If the context is `Teacher`, you could navigate to the association class with the expression:

```
self.room
```

This expression would result in a collection of all the room assignments for all the courses to which the teacher is assigned.

Similarly, you can navigate from an association class to either end of the association. Starting with a context of the `Room` class, the following expressions are valid:

```
self.instructor
self.courses
```

Navigating from an association class always results in a single object.

Operations on Types

In OCL, types themselves have predefined operations that can be performed on them. Operations are defined on both value types, and on collections. If the operation is specified on a basic type, use the dot to continue the expression. If the operation is specified on a collection, use the -> operator to continue the expression.

For example, substring is a predefined OCL operation defined for the string type. The following expression returns the number of characters in the `firstName` attribute:

```
self.firstName.substring(start,stop)
```

However, given the expression:

```
self.Courses->size()
```

In this expression, `self.Courses` results in a collection of all `Course` objects. The arrow operator is used instead of the dot operator. The expression returns the number of elements in the collection.

Some operations take parameters. When this is the case, the parameter list is enclosed in parentheses. For operations that do not take parameters, such as size, the OCL specification calls for parentheses with an empty argument list. However, in ECO you can omit the empty parentheses.

Operations on Basic Types

The following table shows some of the operations defined by the OCL specification for the basic types.

Type	Operations

`Integer`	`=, +, -, /, *`
	`abs()`
	`div(i: Integer)`
	`mod(i: Integer)`
	`max(i: Integer)`
	`min(i: Integer)`

`Real`	`=, <>, <, >, <=, >=`
	`+, -, /, *`
	`abs()`
	`floor()`
	`round()`
	`max(r: Real)`
	`min(r: Real)`

Boolean	`=` `or, xor, and, not`

String	`=`
	`length()`
	
	Note: The OCL specification defines the `size()` operation on strings. ECO uses the name `length()`.
	
	`toUpper()`
	`toLower()`
	`subString(low: Integer, high: Integer)`
	`concat(s: String)`
	
	Note: In ECO, the operator `+` is defined as a string concatenator.

For all types, the operators +, –, *, /<, >, <>, <=, >=, div, mod, and, or, xor, may be written using infix notation. The following two expressions are equivalent:

```
a.+(b)
a + b
```

Perhaps the most common operation performed on a type is the allInstances operation. The allInstances operation retrieves a collection of all instances of the given type. For example, given the expression:

```
Teacher.allInstances()
```

This expression results in a collection of all `Teacher` objects in the ECO space.

Operations on Meta Types

The following table shows other operations that are defined for all types.

Operation	Result
`typeName`	Returns the name of the type.
`attributes`	Returns the set of attributes of the type.
`associationEnds`	Returns the set of navigable association ends.
`supertypes`	Returns the set of all direct supertypes.
`allSuperTypes`	Returns the entire set of supertypes.
`allSubClasses`	Returns the set of all subclasses defined on the type.

Other Type-related Operations

The following table shows some miscellaneous operations on types.

Operation	Result
`oclIsKindOf(aType)`	Returns true if the value is of the specified type or one of its subtypes.
`oclIsTypeOf(aType)`	Returns true if the value is of the specified type exactly.
`oclAsType(aType)`	Returns the same value, but typed as the specified type. A runtime exception is thrown if the typecast fails.

Operations on Collections

The following table shows some of the common operations performed on collections. It is not an exhaustive list.

Operation	Result
`size()`	Returns the number of elements in the collection.
`includes(object)`	Returns `True` if the collection contains the given `object`.
`excludes(object)`	Returns `True` if the collection does not include the given `object`.
`count(object)`	Returns the number of times `object` occurs in the collection.
`isEmpty()`	Returns `True` if the collection is empty.
`notEmpty()`	Returns `True` if the collection is not empty.

Iterators

There is a special construct in OCL called iterators. Iterators are defined for all collections, and behave different from normal operations. For example:

```
Teacher.allInstances()->select(courses->size() > 2)
```

he above expression will iterate over all the teacher objects, and return a new collection with the teachers that fulfill the condition in the select statement. Normally in an expression it is possible to omit the self keyword since this is implicit. Inside an iteration the implicit variable is the loop-variable. The `courses` in the example will be applied to an implicit variable of the type `Teacher` (regardless of the context of the expression). The following table shows the various iterators in OCL.

Iterator	Result
`select(booleanexpr)`	Returns the collection of elements that yields `True`.
`reject(booleanexpr)`	Returns the collection of elements that yields `False`.
`orderBy(anyexpr)`	Returns the same set of elements but ordered according to the `anyexpr`.
`orderDescending(anyexpr)`	Reverse order compared to previous.
`forAll(booleanexpr)`	Returns `True` if all of the objects in the collection yields `True`.
`exists(booleanexpr)`	Returns `True` if one of the object in the collection yields `True`.
`collect`	Returns a collection if the values returned by the iteration expression.
`iterate`	Generic iteration in the OCL specification , but not implemented in ECO.

Iterators can be nested as in this example:

```
Teacher.allInstances()->select(courses->exists(numberOfStudents > 2))
```

It is also possible to make the implicit iterator variable explicit:

```
Teacher.allInstances()->select(t | t.courses->size() > 2)
```

Here the variable `t` is introduced and used to reference the loop variable.

Using ECO Action Language

ECO Action Language is an extension of the Object Constraint Language (OCL). ECO Action Language allows all the operations of OCL, with simple enhancements that produce side effects in the model. ECO Action Language is used primarily on ECO state machine diagrams, where it is used to express entry and exit activities for states, and the effect activity of state transitions.

ECO Action Language Operations

The following table shows the extensions provided by ECO Action Language.

Operation	Purpose	Example
`:=`	Assignment operator	`self.FirstName := home.owners->first.FirstName`
`;`	Statement separator. The `;` is not a statement terminator. A trailing `;` results in a syntax error.	`self.FirstName := home.owners->first.FirstName;` `self.LastName := home.owners->first.LastName`

`create`
 	
`delete`
	Object creation and deletion	`p := Person.create`
		
		`p.delete`

`let <variable>=<expression> in <expression>`	The `let` operation introduces a variable with an initial value. The variable is then used in one or more statements following the `in` keyword. When multiple statements follow the `in` keyword, they are grouped in parentheses. The statements must be separated with the statement separator, `;`.	`let p=Person.create in (p.FirstName='Peter'; p.LastName='Jones')`

ECO Action Language supports the following operations on lists:
`clear`
`add`
`remove`
`removeAt`
	List operations use the corresponding operation on the `IElementList` interface.	`aPerson.ownedBuildings.clear`
		`aPerson.ownedBuildings.add(home)`
		`aPerson.ownedBuildings.remove(home)`
		`aPerson.ownedBuildings.removeAt(3)`

The ECO Action Language also allows calling methods (including triggers) defined on a class. To be callable by ECO Action Language, each parameter of the method must be a value type, an instance of an ECO class, or an array of either of these.

Note: The `let` operation does not have side effects. It is an ECO extension to OCL, and is available in the ECO Action Language**Expression Editor**.

Using ECO Action Language

Like the **OCL Expression Editor**, the ECO Action Language Expression Editor is a property editor that is accessed from the **Object Inspector**. The following table shows where ECO Action Language is available in the IDE:

Property	IDE Location
Entry and Exit actions of states	ECO state machine diagrams. Select the state and access the **Action Language Editor** from the **Object Inspector**.
Effect action of state transitions	ECO state machine diagrams. Select the state transition and access the **Action Language Editor** from the **Object Inspector**.
Method body property of class methods	ECO class diagrams. Select the class and access the **Action Language Editor** from the **Object Inspector**.
Action Expression property of button controls	Select the button on the **Windows Form designer** and access the **Action Language Editor** from the **Object Inspector**.

Using the ECO framework with Multi-Client Applications

These topics describe using the ECO framework in multi-client environments such as ASP.NET.

In This Section

The ECO framework and ASP.NET
Describes basic concepts required for understanding how to build an ECO ASP.NET application.

Using the ECO Framework in Multi-Client Applications
Describes the concepts and components used when writing multi-client ECO framework applications.

The ECO framework and ASP.NET

This topic describes the fundamental concepts of using the ECO framework with ASP.NET applications. You need to have a good understanding of ASP.NET applications and ECO framework applications to fully understand the content of this topic.

Requests, Sessions and Applications

To understand how the ECO framework works with ASP.NET applications, you must have a good understanding of the ASP.NET concepts of requests, sessions, and applications:

- A request can be either a page production or a web service call.
- A session encompasses a series of requests.
- An application is essentially a directory of related .aspx files.

The classes that serve each request are stateless. However, it is possible to cache state on the server. Caching can be done either per session or for the entire application. The ECO framework and ASP.NET use both strategies.

The ECO Space Provider

The class that manages creation and releasing an ECO space is called `EcoSpaceProvider`. `ECOSpaceProvider` is an abstract class and cannot be instantiated. It is declared for you by the **ECO ASP.NET Web Application** and the **ECO ASP.NET Web Service** wizard. `EcoSpaceProvider` contains a property that sets the session caching policy, and static methods to get and release an ECO space.

There is one `EcoSpaceProvider` class for an ECO ASP.NET application. The initial web page and all subsequent web pages created by the **ECO ASP.NET Page** wizard contain an `EcoSpace` property that uses the `EcoSpaceProvider` of the application.

Creating a new ECO space for each request would be expensive, so the ECO framework provides options. The options for the session state caching policy are set by modifying the following line of code, which you will find in the EcoSpaceProvider source file:

```
[C#]
private const EcoSpaceStrategyHandler.SessionStateMode sessionStateMode =
EcoSpaceStrategyHandler.SessionStateMode.Always;
```

```
[Delphi]
const
   MODE: EcoSpaceStrategyHandler.SessionStateMode =
EcoSpaceStrategyHandler.SessionStateMode.Always;
```

The following list shows the possible values and their meaning.

Value	Meaning
`Never`	Never cache the ECO space in the session state. Instead, return it to the pool. If pooling is not used, release it. Any unsaved changes to the ECO space will be lost.
`Always`	Keep a private ECO space in the session state for the duration of the session. When the session ends, the ECO space will be returned to the pool. If pooling is not used it will be discarded. The ECO space will always contain the objects used by the session, which may be more efficient than getting one from the pool that could have different contents, but the disadvantage is that it ties up resources much longer.
`IfDirty`	Keep the ECO space in the session state if it contains dirty objects.

This mode allows applications to keep state over multiple requests.

Keeping ECO spaces in the session state ties up resources on the server. It is more efficient to write applications to be stateless. Stateless applications can set `SessionStateMode` to `Never`.

Note: The code generated by the template for an ECO Web Service application does not use the session state mode property. Instead, ECO web service applications use the methods `EcoSpaceProvider.GetSessionFreeEcoSpace` and `EcoSpaceProvider.ReturnSessionFreeEcoSpace`. These methods ensure that the web service is always stateless.

ECO Space Pooling

You can configure the `EcoSpaceProvider` to maintain an application-wide pool of ECO spaces. ECO spaces are then drawn from and returned to the pool. Each time an ECO space is drawn from the pool, the contents are synchronized with a call to the Sync method of the IPersistenceService interface.

To maintain a pool, each ECO space must share the same PersistenceMapper. The PersistenceMapper can be local (through a PersistenceMapperSharer component), or remote (through a PersistenceMapperClient component).

While the session state strategy affects the semantics of the entire application, ECO space pooling is a deployment option. Pooling is controlled by two settings in the webconfig file.

Setting	Description
`MaxPool`	Setting `MaxPool` to an integer greater than zero will enable pooling of ECO spaces.
`MaxAge`	Pooled ECO spaces will be discarded when they reach `MaxAge` seconds.

The following example webconfig file shows these two settings:

```
<!--  Application settings -->
<appSettings>
    <add key = "Borland.Eco.Web.MaxPool" value = "0" />
    <add key = "Borland.Eco.Web.MaxAge" value = "600" />
</appSettings>
```

Using the ECO Framework in Multi-Client Applications

The ECO framework can synchronize multiple ECO spaces connected to the same database. The changes written to the database by one ECO space can be applied to the others. You need to be familiar with the ECO framework, and with multi-client application technologies such as ASP.NET, and .NET Remoting to fully understand the content of this topic.

This topic introduces the following concepts:

- Shared persistence mappers
- Sharing ECO spaces in-process and across process boundaries
- Synchronization and conflict resolution

Shared Persistence Mappers

All ECO spaces that are to be synchronized must be connected to the same PersistenceMapper. The PersistenceMapper components are both thread-safe and remotable. Set up your application for sharing persistence mappers by adding a `PersistenceMapperProvider` (**File** ▸ **New** ▸ **Other**) and then adding components from the **Tool Palette** to the ECO space.

The `PersistenceMapperProvider` component binds together a PersistenceMapper and its related components, such as a BdpConnection component. The `PersistenceMapperProvider` component has its own design surface. It is similar to the **ECO space designer**, but it only allows database creation and evolution. At runtime, the `PersistenceMapperProvider` holds a single instance of the PersistenceMapper, so it can be shared by multiple ECO spaces.

Note: Synchronization is only supported for the database persistence mapper components, PersistenceMapperSqlServer, and PersistenceMapperBdp. Synchronization is not supported with the PersistenceMapperXML component.

There are two ways to connect the ECO space to a `PersistenceMapperProvider`. The method you use depends on whether all ECO spaces will be running in a single process or in multiple processes.

Using a Shared Persistence Mapper in a Single Process

ASP.NET applications that are deployed on a single server, and Windows Forms applications that create multiple ECO space instances can share a persistence mapper in a single process. Place a PersistenceMapperSharer component on the **ECO space designer**, and connect its MapperProviderType property to the `PersistenceMapperProvider`.

Using a Shared Persistence Mapper in a Separate Process

The `PersistenceMapperProvider` component can be shared using standard .NET Remoting. To share ECO spaces over process boundaries you use a PersistenceMapperClient on the **ECO Space Designer**.

The template for the `PersistenceMapperProvider` component generates a block of sample code that sets up the remoting parameters. This code is initially commented out. You can uncomment the code and make adjustments as necessary.

Sychronizing ECO Spaces

ECO spaces can be synchronized after they are configured to share a `PersistenceMapper`.

Synchronize ECO spaces by calling methods on the IPersistenceService interface. The simplest case is to call the Sync method. When using the ECO framework with ASP.NET, Sync is automatically called whenever an ECO space

is retrieved from the pool. Sync always succeeds if the ECO space is clean. This is always the case for pooled ECO spaces under ASP.NET.

Synchronization and Conflict Resolution

Conflicts can occur when synchronizing if optimistic locking fails or as a side-effect of reading.

An ECO space maintains an old value for all elements that have been fetched. The old value is the value that was read from the database when the object was last fetched. For items that are modified, the ECO space maintains the new value. A modification can be either creating or deleting an object, changing the value of an attribute, or modifying an association.

A conflict occurs if the value in the database is different from the value maintained by the ECO space. Each time a conflict occurs, it is registered in an internal list maintained by the ECO space. This list can be retrieved using the IPersistenceService interface.

The call to RetrieveChanges will query the PersistenceMapper for any changes that might have occurred in other ECO spaces, and record them as potential conflicts.

The list of all unresolved conflicts can be retrieved by calling the GetChanges method of the IPersistenceService interface. GetChanges returns an array of IChange interface instances. The conflicts are resolved by looping over each change in code, and marking the desired action in the Action property of the IChange interface. The Apply method of the IChange interface can be called for each item in the list, or, the IPersistenceService method ApplyChanges can be called to apply the actions all at once. For each change, the following actions are available

Action	Meaning
`Ignore`	No action will be performed; the change will be removed from the list. This action should only be taken if you have handled the conflict in some other manner. For example, by directly changing the object.
`Discard`	The value in the ECO space will be marked as invalid. The value will be reread from the database the next time it is accessed. If the value was dirty, the new value will be discarded. This option is semantically identical to `Reread`.
`Reread`	The value is reread from the database. If the value was dirty, the modified value will be discarded.
`Keep`	The ECO space will update its notion of the old value to the one currently in the database, but will keep any modified value. If the value is not modified `Keep` is identical to `Reread`.
`Verify`	Verify that the potential conflict is a conflict by reading the value from the database. This can happen if another user has modified parts of an object that are not loaded in this ECO space, or if the ECO space has lost and reestablished contact with the persistence server. In this case all loaded objects will be marked as potential conflicts.
`Undecided`	No action is performed, and the change is left in the list. This is the action set on any new changes discovered.

The Sync method of the IPersistenceService interface will resolve all changes where it is safe, for example, where the element in question has not been modified. The example code below shows how a method like Sync could be implemented.

```
[C#]
public void MySync(bool ReadNewValues)
{
  IChange[] Changes;
  Change: IChange;
```

```
  EcoSpace.PersistenceService.RetrieveChanges();
  Changes = EcoSpace.PersistenceService.GetChanges();
  foreach Change in Changes {
    if(!Change.IsDirectConflict) {
      if(ReadNewValues)
        Change.Action = ChangeActionKind.Reread;
      else
        Change.Action = ChangeActionKind.Discard;
    }
  }
  EcoSpace.PersistenceService.ApplyAll();
}
```

```
[Delphi]
procedure WebForm1.MySync(ReadNewValues: Boolean);
var
  Changes: ChangeArray;
  Change: IChange;
begin
  EcoSpace.PersistenceService.RetrieveChanges;
  Changes := EcoSpace.PersistenceService.GetChanges;
  for Change in Changes do
    if not Change.IsDirectConflict then
      if ReadNewValues then
        Change.Action := ChangeActionKind.Reread
      else
        Change.Action := ChangeActionKind.Discard;
  EcoSpace.PersistenceService.ApplyAll();
end;
```

Custom OR Mapping

These topics discuss the format of the ECO framework custom OR mapping file.

In This Section

Custom ECO Object-Relational Mapping Files
Describes the format of the object-relational mapping file used by the ECO framework.

Custom OR Mapping with Auto-Increment Columns
Describes an example of a custom OR mapping file that includes an auto-increment column.

Custom OR Mapping with BLOB Tables
Describes an example of a custom OR mapping file that stores BLOBs in the database.

Custom OR Mapping with Objects stored in multiple tables, with multiple keys
Describes an example of a custom OR mapping file that stores objects in multiple tables, and implements multiple keys.

Custom OR Mapping with Singlelink and Compound Keys
Describes an example of a custom OR mapping file that includes a singlelink definition, and a class with a compound key.

Custom OR Mapping Using Type Discriminator Columns
Describes an example of a custom OR mapping file that uses a type discriminator column.

Custom ECO Object-Relational Mapping Files

An object-relational (OR) mapping specifies how to map classes and relationships defined in the model to a relational database schema. All models created with the ECO framework have a default OR mapping. The **ECO Space designer** can generate and evolve a database schema specified by this default OR mapping.

If you work entirely within the IDE, using the class diagram surface to develop the model, creating and evolving the database schema using the **ECO Space designer**, then you will never have to work with custom OR mapping files. There might be situations where you want to develop the database schema yourself. Or, you might be working under the restrictions imposed by a database administrator. The most common case is when you have an existing database that you would like to use to build a model.

The ECO framework is capable of reverse engineering an existing database. The outcome of reverse engineering is a set of classes with attributes and associations (contained within a single model package file), and a custom OR mapping file, which is specified in XML.

This topic discusses the following:

- Custom OR mapping file format.
- Using custom OR mapping files with database schema evolution.

Custom OR Mapping File Format

You can produce a custom OR mapping file manually, or you can use the **ECO Space Designer** to reverse engineer an existing schema. Reverse engineering a database is a complex procedure. The correct object-oriented classes and relationships cannot always be inferred from the schema, and you might need to make manual modifications to the XML mapping file.

XML Mapping File Specification

The following shows the specification of the XML file produced by the reverse engineering tools. The persistence mapper components used by the ECO framework require the custom OR mapping file to adhere to this specification.

```
<Globals [ImplicitAliasInFeatures:bool] [ImplicitColumnInFeatures:bool]>?
</Globals>
<ORMapping>1
  <ClassDef Name:string>+
    <AliasDef Name:string Table:string [IsMainAlias:bool]
              [ExtentRequiresDiscriminator:bool]>*
      <KeyImpl Name:string [IsAutoInc:bool]>+
        <KeyColumn Name:string />+
        <DiscriminatorColumn Name:string />*
        <ConstantColumn Name:string Signature:string value:string/>*
      </KeyImpl>
      <DiscriminatorImpl Name:string Column:string/>
    </AliasDef>
    <KeyDef Name:string Signature:string [IsId:bool] KeyMapper:string/>*
    <DiscriminatorDef Name:string Signature:string />*
    <DiscriminatorValue: Name:string Value:string IsFinal:bool/>*
    <AttributeDef Name:string [Columns:string] [Alias:string] [AllowNULL:bool]
          [Length:int] />*
    <SingleLinkDef Name:string [Columns:string] [OrderColumn:string] [Alias:string]
           Key:string [IsConstrained:bool] />*
  </ClassDef>
  <Database Name:string>*
    <Table Name:string>+
      <Column Name:string AllowNULL:bool Type:string Length:int
```

```
              DefaultValue:string/>+
      <Index Name:string Columns:string IsUnique:bool IsCaseSensitive:bool IsPrimary:bool
              IsDescending:bool/>*
    </Table>
  </Database>
</ORMapping>
```

Legend:

- “[ ]” indicates that an attribute is optional
- “+” means that a node must occur at least once
- “*” means that a node can occur zero or more times
- “?” means that a node is optional
- “1” means that a node must occur exactly once

XML Elements of Custom OR Mapping Files

From the specification, you can see that the mapping file defines each class in the model. There will be one <ClassDef> node for each persistent class. The attributes and subnodes of <ClassDef> determine how instances of the class are to be stored in the database.

The following are the individual elements of the OR mapping file:

Element	Description
<Globals> Element	Defines global attributes that apply to the entire mapping file.
<Classes> Element	A top level node that groups all the classes defined in the OR mapping file.
<ClassDef> Element	The storage characteristics of each class are contained within a `<ClassDef>` element and its subnodes.
<AliasDef> Element	For each class, you must specify a reference to the table or tables that will store instances of the class. These table references are contained in `<AliasDef>` elements. Multiple aliases can refer to the same table, if the same table stores instances of multiple classes.
<KeyDef> Element <KeyImpl> Element <KeyColumn> Element	The keys that exist to identify each instance of the class are specified in `<KeyImpl>`, `<KeyColumn>` and `<KeyDef>` nodes. Each `<ClassDef>` can contain multiple keys, but exactly one of them must specify a key to be the ID. The ID key is the unique identity of an object in the object layer. It is normally also the primary key of the object in the database.
<ConstantColumn> Element	A constant column is used if a column should always have the same value.
<DiscriminatorDef> Element <DiscriminatorImpl> Element <DiscriminatorValue> Element <DiscriminatorColumn> Element	If a type discriminator is required, it is specified with `<DiscriminatorImpl>`, `<DiscriminatorValue>`, `<DiscriminatorColumn>`, and `<DiscriminatorDef>` elements. Type discriminators are required if, for example, a class has any subclasses, or if unrelated classes stored in the same table.
<AttributeDef> Element	Each attribute of the class is specified in `<AttributeDef>` elements.
<SingleLinkDef> Element	The `<SingleLinkDef>` element must be specified when the model requires that a class has a singlelink relationship to another class.
<Database> Element	In most cases, ECO will determine the database schema automatically if you specify the `<ClassDef>` nodes. In some cases, the `<Classdef>` nodes are not suffient, so it is possible to explicitly specify the schema of the database with the `<Database>` element.

	This is only required if the schema is to be generated in an empty database. For normal data operations, this is never required. There are properties that can be specified in both the `<ClassDef>` section and the `<Database>` section: Attribute.Length / Column.Length, and Attribute.AllowNULL/Column,AllowNULL. The setting in the `<Database>` section will override the setting in the `<ClassDef>` section.

The `<AliasDef>`, `<KeyDef>`, `<DiscriminatorDef>`, `<AttributeDef>`, and `<SingleLinkDef>` elements are all inherited by subclasses in the custom OR mapping file. If a `<ClassDef>` element specifies a superclass, that `<ClassDef>` will inherit all of the elements in its superclass.

Classes Stored in Multiple Tables

If a class is stored in more than one table, its `<ClassDef>` will have more than one `<AliasDef>` subnode defined in the mapping file. Also note that the set of aliases for a class includes its own, and any aliases that are and inherited from other classes.

It must be possible to “join” all aliases of a class (including the inherited aliases). This means that each alias must have a `<KeyImpl>` node that refers to a key definition that is also present in some other alias. At least one alias must implement a key that is marked with `IsId="true"`.

Any alias that does not implement the ID key (the IsId attribute is not set to `"true"`) must implement some key that is implemented by one of other aliases.

Custom Object-Relational Map Files and Database Evolution

When using a custom mapping file, you must link the persistence mapper component on the ECO space to the mapping file.

The built-in database evolution mechanism on the **ECO Space designer** must be able to find the old mapping file (the mapping file prior to alterations) and the new mapping file.

Custom OR Mapping with Auto-Increment Columns

The following example describes a custom OR mapping file that includes an auto-increment column. Auto-increment columns get their value from the database itself. The value generated by the database is guaranteed to be unique within the table.

This example includes one class, called `Product`. The class has an attribute called `ProdNum` that stores its value in an auto-increment column. The `ProdNum` column is also used as a key.

```
<ClassDef Name="Product">
  <AliasDef Name="Product_ALIAS" Table="Product" >
    <KeyImpl Name="ProdKey" IsAutoInc="true">
      <KeyColumn Name="ProdNum" />
    </KeyImpl>
  </AliasDef>
  <KeyDef Name="ProdKey" Signature="System.Int32" IsId="true"
          KeyMapper="AutoInc" />
  <AttributeDef Name="ProductNumber" Alias="Product_ALIAS" Columns="ProdNum"
          AllowNULL="False"/>
  <AttributeDef Name="Name" Alias="Product_ALIAS" Columns="Name"
          AllowNULL="False"/>
  <AttributeDef Name="Price" Alias="Product_ALIAS" Columns="Price"
          AllowNULL="False"/>
  <AttributeDef Name="Color" Alias="Product_ALIAS" Columns="Color"
          AllowNULL="False"/>
  </ClassDef>
```

Notice the `<KeyImpl>` node sets the `IsAutoInc` attribute to "true", indicating the table column used for the key is an auto-increment column.

Custom OR Mapping with BLOB Tables

The following example describes a custom OR mapping file that stores BLOBs in the database. BLOBs, or Binary Large Objects, consist of purely binary data, as the name implies. BLOBs are typically used to store images or sounds in the database.

This example shows a class with two BLOB attributes. Some databases only allow a table to have a single BLOB field, but with this mapping, you can store both BLOBs (or any BLOB in the model) in the same table.

First, define a main `<AliasDef>` in which to store the ImageId attribute and the Description. This alias is connected to the `ImageInfo` table. Next, define two new aliases, both connected to the `ImageData` table. Both aliases use the `ImageId` key. In addition to the KeyColumn, the aliases also specify a ConstantColumn called `ImageType`, with different values (one value for the thumbnail and one for the full size photo). The `ImageData` table will have three columns: `ImageId`, `ImageType` and `BlobData`.

```
<ClassDef Name="Image”>
  <AliasDef Name="Image_Main" Table="ImageInfo" IsMainAlias="true"
            ExtentRequiresDiscriminator="false">
    <KeyImpl Name="ImageId">
      <KeyColumn Name="ImageId" />
    </KeyImpl>
  </AliasDef>
  <AliasDef Name="ThumbAlias" Table="ImageData"
            ExtentRequiresDiscriminator="false">
    <KeyImpl Name="ImageId">
      <KeyColumn Name="ImageId" />
      <ConstantColumn Name="ImageType" Signature="System.String"
          Value=”Thumb” />
    </KeyImpl>
  </AliasDef>
  <AliasDef Name=" PhotoAlias" Table="ImageData"
            ExtentRequiresDiscriminator="false">
    <KeyImpl Name="ImageId">
      <KeyColumn Name="ImageId" />
      <ConstantColumn Name="ImageType" Signature="System.String"
                      Value="Photo" />
    </KeyImpl>
  </AliasDef>
  <KeyDef Name="ImageId" Signature="System.Int32" IsId="true"
          KeyMapper="Attribute">
  <AttributeDef Name="ImageId" Alias="Image_Main" Columns="ImageId"
          AllowNULL="False"/>
  <AttributeDef Name="Description" Alias="Image_Main" Columns="Desc"
          AllowNULL="False" Length=255/>
  <AttributeDef Name="Photo" Alias="PhotoAlias" Columns="BlobData"
          AllowNULL="False"/>
  <AttributeDef Name="Thumb" Alias="ThumbAlias" Columns="BlobData"
          AllowNULL="False"/>
</ClassDef>
```

Custom OR Mapping with Objects stored in multiple tables, with multiple keys

The following example describes a custom OR mapping file that stores objects in multiple tables, and implements multiple keys.

This example shows how secondary tables can be joined either using the key that is marked as “IsId”, or by a secondary key. The first `<AliasDef>` element implements both keys that are defined for the class `Invoice`. The second `<AliasDef>` implements the ID key, and the third implements the secondary key.

It must be possible to link all of the aliases into one *graph* by pairing them together with shared key implementations.

```
<ClassDef Name="Invoice">
  <AliasDef Name="Invoice_ALIAS" Table="Invoice" IsMainAlias="True">
    <KeyImpl Name="InvoiceKey">
      <KeyColumn Name="InvoiceNumber" />
    </KeyImpl>
    <KeyImpl Name="SecondKey">
      <KeyColumn Name="SecondKey" />
    </KeyImpl>
  </AliasDef>
  <AliasDef Name="Invoice2_ALIAS" Table="Invoice_IdJoin">
    <KeyImpl Name="InvoiceKey">
      <KeyColumn Name="InvoiceNumber" />
    </KeyImpl>
  </AliasDef>
  <AliasDef Name="Invoice3_ALIAS" Table="Invoice_KeyJoin">
    <KeyImpl Name="SecondKey">
      <KeyColumn Name="SK" />
    </KeyImpl>
  </AliasDef>
  <KeyDef Name="InvoiceKey" Signature="System.Int32" IsId="true"
          KeyMapper="Attribute" />
  <KeyDef Name="SecondKey" Signature="System.Int32"
          KeyMapper="Attribute"/>
  <AttributeDef Name="InvoiceNumber" Alias="Invoice_ALIAS" Columns="InvoiceNumber"
          AllowNULL="False”/>
  <AttributeDef Name="MainAttr" Alias="Invoice_ALIAS" Columns="MainAttr"
          AllowNULL="False"/>
  <AttributeDef Name="Attr2" Alias="Invoice2_ALIAS" Columns="Attr2"
          AllowNULL="False"/>
  <AttributeDef Name="Attr3" Alias="Invoice3_ALIAS" Columns="Attr3"
          AllowNULL="False"/>
  <AttributeDef Name="SecondKey" Alias="Invoice_ALIAS" Columns="SecondKey"
          AllowNULL="False"/>
</ClassDef>
```

Custom OR Mapping with Singlelink and Compound Keys

The following example describes a custom OR mapping file that includes a singlelink definition, and a class with a compound key.

In this example there are two classes: `CEO` and `Company`. The CEO class has two attributes, `FirstName` and `LastName`, that are used as a compound key in the database. The two classes are related such that a `Company` has exactly one `CEO`.

The XML required to define this relationship in the mapping file is shown below:

```
<ClassDef Name="CEO">
  <AliasDef Name="CEO_ALIAS" Table="CEO"
            ExtentRequiresDiscriminator="False">
    <KeyImpl Name=" CEOID">
      <KeyColumn Name="FName" />
      <KeyColumn Name="LName" />
    </KeyImpl>
  </AliasDef>
  <KeyDef Name="CEOID" Signature="System.String, System.String" IsId="true"
          KeyMapper="Attribute">
  <AttributeDef Name="FirstName" Alias="CEO_Alias" Columns="FName" AllowNULL="False"
          Length="10" />
  <AttributeDef Name="LastName" Alias="CEO_Alias" Columns="LName" AllowNULL="False"
          Length="10" />
</ClassDef>

<ClassDef Name="Company">
  <AliasDef Name="Company_Alias" Table="Company">
    <KeyImpl Name="CompanyId">
      <KeyColumn Name="CompId" />
    </KeyImpl>
  </AliasDef>
  <KeyDef Name="CompanyIdKey" Signature="System.Int32" IsId="true"
          KeyMapper="Attribute">
        <AttributeDef Name="CompanyId" Alias="Company_Alias" Columns="CompId"
          AllowNULL="False"/>
  <AttributeDef Name="Name" Alias="Company_Alias" Columns="Name" AllowNULL="False"
          Length="255" />
  <SingleLinkDef Name="CEO" Alias="Company_Alias" Columns="CEOFName,CEOLName"
          Key="CEO.CEOID" />
</ClassDef>
```

The compound key is specified in the `<KeyDef>` node of the `CEO` `<ClassDef>` element. The key implementation (`<KeyImpl>`), shows that the key contains two columns, `FName`, and `LName`. These refer to column names in the database.

The singlelink relationship is defined in the `Company` `<ClassDef>` element. The association's end is named "CEO". The table columns that store the CEO data are `CEOFName` and `CEOLName`. The key is specified in the `CEOID` `<KeyDef>` element, in the CEO `<ClassDef>`.

Custom OR Mapping Using Type Discriminator Columns

The following example describes a custom OR mapping file that uses a type discriminator column. Type discriminator columns are required when a single table is used to store multiple classes. The type discriminator value is used to distinguish one class from another.

In this example, two classes (`Male` and `Female`) are mapped to the same table (the `Person` table). An abstract root class has been added to define the discriminator (it could also have been defined twice, once on each class, like the ID key). Since the two classes are located in the same table, a discriminator is required to find the extent (i. e. all instances) of the different classes.

By specifying the `<AliasDef>` attribute `ExtentRequiresDiscriminator="true"`, the SQL for finding all objects of type `Male` will change from

```
SELECT Id FROM Person
```

to

```
SELECT Id FROM Person WHERE Sex = 'M'
```

```
<ClassDef Name="RootClass">
  <DiscriminatorDef Name="Sex" Signature="System.Char">
  <DiscriminatorValue Name="Sex" Value="X">
    </ClassDef>

<ClassDef Name="Male" SuperClass="RootClass">
  <AliasDef Name="MaleAlias" Table="Person"
            ExtentRequiresDiscriminator="true">
    <KeyImpl Name="Id" IsAutoInc="true">
      <KeyColumn Name="Id" />
    </KeyImpl>
    <DiscriminatorImpl Name="Sex" Column="Sex">
  </AliasDef>
  <KeyDef Name="Id" Signature="System.Int32" IsId="true"
          KeyMapper="AutoInc" />
  <DiscriminatorValue Name="Sex" Value="M">
  <AttributeDef Name="Name" Alias="MaleAlias" Columns="Name"
                AllowNULL="False"/>
  <AttributeDef Name="Age" Alias="MaleAlias" Columns="Age"
                AllowNULL="False"/>
</ClassDef>

<ClassDef Name="Female" SuperClass="RootClass">
  <AliasDef Name="FemaleAlias" Table="Person"
            ExtentRequiresDiscriminator="true">
    <KeyImpl Name="Id" IsAutoInc="true">
      <KeyColumn Name="Id" />
    </KeyImpl>
    <DiscriminatorImpl Name="Sex" Column="Sex">
  </AliasDef>
  <KeyDef Name="Id" Signature="System.Int32" IsId="true"
          KeyMapper="AutoInc" />
  <DiscriminatorValue Name="Sex" Value="F">
  <AttributeDef Name="Name" Alias="FemaleAlias" Columns="Name"
                AllowNULL="False"/>
```

```
  <AttributeDef Name="Age" Alias="FemaleAlias" Columns="Age"
                AllowNULL="False"/>
</ClassDef>
```

Building Web Applications with ASP.NET

ASP.NET is the programming model for building Web applications using the .NET Framework. This section provides the conceptual background for building ASP.NET applications using Developer Studio 2006. In addition to supporting data access components within the .NET Framework, Developer Studio 2006 includes DB Web Controls. DB Web Controls work with .NET Framework providers and Borland Data Providers for .NET (BDP.NET) to accelerate Web application development.

In This Section

ASP.NET Overview
Introduces the ASP.NET architecture for building Web applications using Developer Studio 2006.

Building an ASP.NET Application
Describes the essential tasks for creating an ASP.NET application.

Borland DB Web Controls Overview
Introduces Borland DB Web Controls, a component framework for rapid application development in ASP.NET using Borland data-aware controls.

Working with DataViews
Describes the DataView and issues regarding its use.

Using DB Web Controls in Master-Detail Applications
Covers details pertaining to master-detail relationships when using web controls in ASP.NET.

DB Web Controls Navigation API Overview
Describes Web DB Controls Navigation API.

DB Web Control Wizard Overview
Describes what you need to know to create DB Web Controls.

Using XML Files with DB Web Controls
Describes how to use XML files as a data source for DBWeb Controls.

Working with WebDataLink Interfaces
Describes the interfaces implemented by the DB Web Controls.

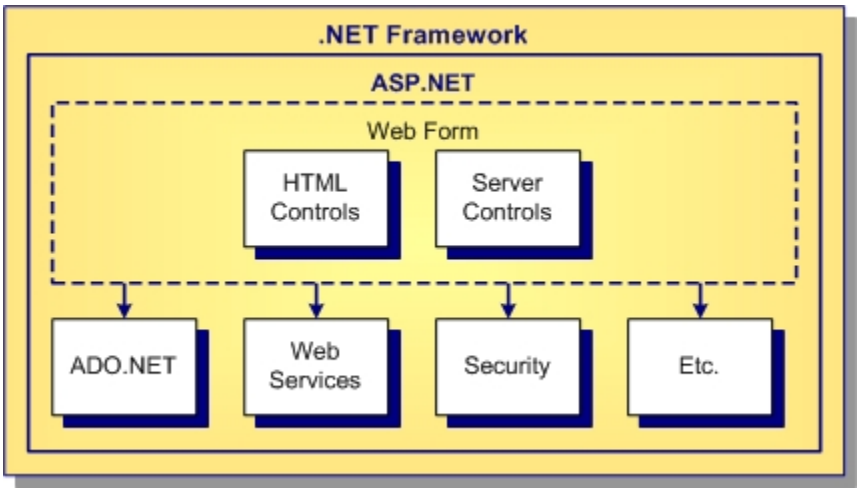
Deploying ASP.NET Applications
Provides general information about deploying ASP.NET applications.

ASP.NET Overview

ASP.NET is the .NET programming environment for building applications in HTML that run on the Web. This topic provides introductory information about the major components of the ASP.NET architecture and explains how ASP.NET integrates with other programming models in the .NET framework. This topic introduces:

- ASP.NET Architecture
- Web Forms
- Data Access
- Web Services
- Designtime Features
- Supported Web Servers
- Sample Applications

ASP.NET Architecture



The major components of the ASP.NET architecture are Web Forms, ASP.NET server controls, code-behind logic files, and compiled DLL files. Web Form pages contain HTML elements, text, and server controls. Code-behind files contain application logic for the Forms page. Compiled DLL files render dynamic HTML on the web server.

Borland provides tools to simplify ASP.NET development. If you are familiar with rapid application development (RAD) and object oriented programming (OOP) using properties, methods, and events, you will find the ASP.NET model for building Web applications familiar.

Web Forms, Server Controls, and HTML Elements

Web Forms define the user interface for your Web application. Typically, a Web Form consists of a markup file (.aspx) that provides the visual presentation and a code-behind file (.pas or .cs) that provides the program logic. The code-behind file is compiled to a .dll and deployed to the server with the .aspx file. At runtime, the .aspx is compiled and linked against the code-behind .dll. This enables you to change the .aspx file without recompiling the code-behind file.

The Web Form .aspx file consists of ASP.NET server controls and static HTML elements. Server controls are declared in your code and can be accessed programmatically through properties, methods, and event handlers. They run on the web server and render HTML to send back to the client.

HTML elements are static, client-side controls; they are not, by default, programmatically accessible. However, they are well suited for static text and images on a Web Form.

Data Access

Web Forms access data through ADO.NET. You can connect an ASP.NET application to an ADO.NET data source by using the data components included in the .NET Framework or the Borland Data Provider (BDP.NET) components included with Developer Studio 2006. BDP.NET components connect to a several industry standard databases.

The Borland DB Web Controls are data-aware components that simplify database development tasks. They work with both the BDP.NET and .NET Framework data components. The DB Web Controls provide advanced functionality and include grid, navigator, calendar, combobox, sound, and video components.

Web Services

Web Services provide application components to many distributed systems using XML-based messaging. A web service can be as simple as an XML message updating values in a remote application or can be an integral part of a sophisticated ASP.NET or ADO.NET application. Web Services and ASP.NET share a common .NET infrastructure that allows for seamless integration.

Supported Web Servers

You can use Internet Information Services (IIS) 6 .0 and/or the Cassini web server while developing your ASP.NET applications. IIS is a comprehensive, scalable web server and is included with Windows Server 2003. You can deploy applications to a computer running IIS.

Cassini is a simpler, no cost web server, suitable for local development and testing, but not intended for application deployment. Cassini can be downloaded from http://www.asp.net/Projects/Cassini/Download. Cassini is also distributed with Developer Studio 2006 and available, by default, in the C:\Program Files\Borland\BDS\4.0\Demos \Cassini directory.

You can use both IIS and Cassini on the same computer, provided you configure them to use different ports.

When you create an ASP.NET application, Developer Studio 2006 prompts you to specify the web server and location for the application. You can set the default server and location for new applications, as well as the Cassini location and port, on the **Tools** ▸ **Options** ▸ **ASP.NET** options page.

Designtime Features

Developer Studio 2006 provides several designtime features to help you accelerate the development of Web Forms, HTML, and CSS files.

Editing HTML and CSS Files

Many of the **Code Editor** features are also available when editing HTML and CSS files. Code Completion (CTRL +SPACE) and syntax highlighting are available for HTML and CSS files. Error Insight is available for HTML files and highlights invalid HTML with a wavy red underline. If you position the mouse over the highlighted HTML, a hint window is displayed indicating the probable cause of the error.

When using the visual Designer, the **Tag Editor** is displayed beneath the Designer, enabling you to view and edit the corresponding markup language for the controls on the Designer. To set the **Tag Editor** and other HTML options, choose **Tools** ▸ **Options** ▸ **HTML Designer Options**.

When displaying an HTML page, the internal HTML formatter automatically indents the HTML to improve readability. Alternatively, you can use HTML Tidy, the standard formatting tool from www.w3c.org. You can use HTML Tidy as needed to format the file and check for errors by choosing the **Edit** ▸ **HTML Tidy** menu commands. Alternatively,

you can set it as the default formatter, instead of the internal formatter. You can also define tags that HTML Tidy would otherwise detect as invalid, such as those prefixed with `asp:`. To access the HTML Tidy options, choose **Tools** ▸ **Options** ▸ **HTML Tidy Options**.

The **Structure View** displays a hierarchical tree view of the HTML tags in the active HTML page and is useful for navigating large files. Double-clicking a node in the tree view positions the HTML file to the corresponding tag.

Designer Flow Layout and Grid Layout

When designing a Web Form, you can use either *grid layout* or *flow layout* for the Designer. In grid layout, controls are arranged by absolute position and you can reposition them by dragging them on the form. An optional, visible grid is also available to help you align controls. If you drag a control from the **Tool Palette** onto the Web Form, or if you click the control on the **Tool Palette** and then click Web Form, the control is added using absolute positioning.

In flow layout, controls are arranged top to bottom on the Web Form, and you can reposition them by using the arrow keys. If you double-click a control on the **Tool Palette**, it will be added to the Web Form in flow layout.

The layout for the Web Form is specified in **Object Inspector** by setting the Document PageLayout property, or in the .aspx file with the `<body ms_positioning="GridLayout">` tag.

The layout for an individual control can be changed by using the **Absolute Layout** button on the **HTML Design** toolbar at the top of the Designer.

To permanently change the layout for new files created with Developer Studio 2006, you can edit the page.aspx template file located at, by default, \BDS\4.0\Objrepos\DelphiDotNet.

Sample Applications

Developer Studio 2006 includes several ASP.NET sample applications in the Demos directory (located, by default, at C:\Program Files\Borland\BDS\4.0\Demos). Many of the sample applications include a readme file that explains the application and lists any prerequisites. Before you attempt to open a sample application in the IDE:

- Check for a readme file in the application's directory and follow any set up instructions.
- Create a virtual directory for the sample application to avoid resource cannot be found errors in the browser at runtime (see the procedure listed at the end of this topic).

Borland DB Web Controls Overview

Borland DB Web Controls simplify database development tasks in combination with BDP.NET and .NET Framework data access components. DB Web Controls are data-aware controls that provide advanced functionality, including data-aware grid, navigator, calendar, combobox, and other popular components.

This section introduces:

- DB Web Controls Architecture
- Data-aware Components Advantages
- Supported Data Access Components
- DB Web Controls Namespace
- ASP.NET Application Deployment with DB Web Controls

DB Web Controls Architecture

DB Web Controls are a set of visual and non-visual components that speed up the creation of ASP.NET applications by providing drag-and-drop capabilities along with a powerful data source discovery mechanism. For the most part, DB Web Controls are common GUI web controls for ASP.NET applications. The connector control, the DBWebDataSource control, acts as a data-aware connector between the visual controls and the underlying data source. In other words, the DBWebDataSource control acts as a conduit for the data that is stored in a data source and the controls that display that data on your ASP.NET form. The DBWebDataSource control can reference both .NET Framework ADO.NET and BDP.NET components. For example, the in-memory DataSet that is generated by an ADO.NET adapter (such as the SQLDataAdapter) or by one of the BDP.NET adapters (such as the BDPDataAdapter). Additionally, you can use the DBWebDataSource to link to other types of data source providers, such as text files, arrays, or collections.

Data-Aware Components Advantages

Typically, when you create an ASP.NET application that features controls that expose data from an underlying data source, such as a database, you need to manually configure the binding between the data source and the controls. This means figuring out the syntax and parameters for each control that must be bound to the data source.

The major advantage of using DB Web Controls is that once you have connected one DBWebDataSource control to your data source, all of the DB Web Controls on your ASP.NET page that reference the DBWebDataSource automatically bind to the underlying data source. You do not need to add any code to accomplish the data binding.

DB Web Controls provide the following advantages over standard web controls:

- Eliminates a need to call the WebControl.DataBind method. Normally, each ASP.NET control on the web form requires that you add this call in the Page_Load routine or the control will not display data at runtime.
- Provides a designtime view of the data.
- Posts changes back to the DataSet automatically. Typically, ASP.NET controls require code to post back changes.
- Maintains current row position.
- Manages change and row state automatically. This means that clients from different machines can operate independently, without regard to the server-side state.

In addition to these general advantages, DB Web Controls provide the following specific advantages:

- The DBWebDataSource maintains an ordered list of changes so that the user can undo changes in the order in which they were made.

- The DBWebNavigator control provides navigation capabilities for grids, multiple text controls, and can be extended to standard web controls.
- The DBWebDataGrid provides built-in capabilities for paging with numbers and icons, for adding Edit and Delete columns, and other advanced capabilities. In other words, you no longer need to code these features into your grid control.

Supported Data Access Components

DB Web Controls are compatible with .NET Framework ADO.NET and Borland BDP.NET data access components. Any data source that can be accessed by one of these providers can serve as the underlying data source for the DB Web Controls. In addition, many of the DB Web Controls, like many .NET web controls in general, can access other objects as data sources, such as arrays, collections, and files.

DB Web Controls Namespace

The namespace for DB Web Controls is Borland.Data.Web. By using reflection, you can learn much about the structure of the namespace and the controls. You can add the namespace to your project, then open it in the **Code Editor**. This opens the **Reflection Editor** and gives you a hierarchical view of all of the controls and their members.

Control	Description
DBWebDataSource	Acts as a bridge between the data source and the DBWeb controls.
DBWebAggregateControl	Text box control that displays aggregate values from a specified column.
DBWebCalendar	A calendar control.
DBWebCheckBox	A check box control.
DBWebDropDownList	A combo box control.
DBWebGrid	A data grid.
DBWebImage	An image control.
DBWebLabel	A label.
DBWebLabeledTextBox	A text box with an attached label.
DBWebListBox	A list box control.
DBWebMemo	A memo field control.
DBWebNavigationExtender	A non-visual component that allows you to define standard web control buttons as navigation controls.
DBWebNavigator	A navigation bar.
DBWebRadioButtonList	A radio button group.
DBWebSound	A sound control, which uses the default media player on your system.
DBWebTextBox	A text box.
DBWebVideo	A video control, which uses the default media player on your system.

ASP.NET Application Deployment with DB Web Controls

After creating an ASP.NET project with DB Web Controls, deploy your ASP.NET application as usual. No special considerations are required.

DB Web Controls Navigation API Overview

Although you can use the standard DBWebNavigator control for most applications, you may need to exercise more control over the navigation in your application. The DB Web Controls now provide an API that allows you to fine-tune your navigation. For example, using the API, you can create a button that performs navigation directly, rather than using the standard DBWebNavigator control. Although you can hide buttons on the DBWebNavigator, you might want to place controls in different locations on the form. With DBWebNavigator, for instance, if you hide all buttons but Previous and Next, they still appear side by side. To place the buttons on opposite sides of the form, use the navigation API methods or the DBWebNavigationExtender control. Both allow you to turn standard web control buttons into navigation controls.

To provide this capability, the DBWebDataSource implements new IDBDataSource methods, each of which perform a specific navigation task. You include these methods in the Form_Load event. You are not required to include click events.

The following methods are provided:

- RegisterNextControl
- RegisterPreviousControl
- RegisterFirstControl
- RegisterLastControl
- RegisterInsertControl
- RegisterDeleteControl
- RegisterUpdateControl
- RegisterCancelControl
- RegisterUndoControl
- RegisterUndoAllControl
- RegisterApplyControl
- RegisterRefreshControl
- RegisterGoToControl

Working with DataViews

With DataViews you can set filters on a DataTable using the RowFilter property or place data in a specific order. You can find the DataView component under the **Data Components** area of the **Tool Palette**. This topic discusses:

- Runtime Properties
- Master-Detail Relationships
- ClearSessionChanges Method
- DataView Limitations

Runtime Properties

At designtime, when a DBWeb control points to a DataView, the control is automatically updated whenever there is a change to any DataView property that controls the rows to be displayed. To change the DataView properties at runtime, you must make sure that the change is in place prior to the rendering of any of the DB Web Controls.

For example, if you use a listbox to set the filter, you would also:

- Set the listbox AutoPostback property to **True**.
- Add code in the Page_Load event to handle setting the RowFilter.
- Add code in the Page_Load event to call the ClearSessionChanges method after the RowFilter has been changed.

Assume you have two tables on a form. You bind an ASP.NET listbox to one table that contains lookup values. These values serve as a filter for the second table, whose values display in a DBWebGrid. Set the AutoPostback property in the listbox to **True**, handle the RowFilter setting in Page_Load, and call ClearSessionChanges after changing the RowFilter.

Tip: If you set the AutoRefresh property to **False**, which is the default, you might end up using cached data. Review the WorldTravel demo in `\Demos\DBWeb` to see an example of how this is handled.

Master-Detail Relationships

You can make a DataView the master table in a master-detail relationship by adding a row filter. Set up a master-detail relationship with two or more DataTables within a single DataSet, then connect the DataView to the master DataTable. When the DBWebDataSource connects to the DataView, the DB Web Controls will let you select either the parent table, which is the DataView, or the detail table.

ClearSessionChanges Method

The ClearSessionChanges method notifies the DBWebDataSource that the DataSet has changed and that existing row, column, and changed data information is no longer valid. All pending changes are removed. If you try to call this method from a DBWebNavigator button click event, the DBWebNavigator button will not work.

DataView Limitations

There are some limitations with the DataView:

- Inserted rows in a DataView behave differently than inserted rows in a DataTable.
- A DataView does not allow multiple inserts of null records. This means that you must add data to an inserted row before adding a new inserted row.

- If an inserted row is deleted, that row is removed from the DataView and you cannot use Undo to recall it.
- If an inserted row contains a single non-null value, and that value is set to null, the row can be deleted in some cases and cannot be recalled.
- DBWeb controls do not provide full support for the DataViewSort property. If a sort field is encountered, the values for the fields contained in the Sort property cannot be changed, and the insert key will be disabled on the DBWebNavigator.

Working with WebDataLink Interfaces

The characteristic that makes DB Web Controls different from traditional web controls is that the DB Web Controls automatically handle all data binding for you. Although you must still configure the links between data sources and controls at design time, all runtime binding is handled, without the need for you to add a data binding command in your code. When extending a DBWeb control using the **DBWeb Control Wizard**, you will implement several interfaces that provide the data binding capabilities. These interfaces are discussed in this topic.

- IDBWebDataLink
- IDBWebColumnLink: IDBWebDataLink
- IDBWebLookupColumnLink: IDBWebColumnLink

IDBWebDataLink

All DB Web Controls implement this interface. The interface defines a data source and a data table, allowing you to connect to and access data from a variety of data sources, including databases, text files, arrays, and collections. If your control only needs to access data at the table level, you implement this interface.

IDBWebColumnLink:IDBWebDataLink

This interface is implemented by DBWeb column controls, such as DBWebImage,DBWebTextBox, and DBWebCalendar, among others. The interface defines a column name to which a column control is linked. In combination with the IDBWebDataLink interface, this interface provides access to standard table and column data.

IDBWebLookupColumnLink:IDBWebColumnLink

This interface is implemented by DBWeb lookup controls, such as DBWebListBox,DBWebRadioGroup, and DBWebDropDownList. The interface defines a TableName within a DataSet, a ColumnName representing a table that contains the data to be displayed in the lookup, and the column containing the values which, when a value is selected, are to be placed into the ColumnName field linked to the control. By default, the ColumnName field is the same as DataTextField. Lookup controls contain not only a text property, usually the item that is displayed in the control, such as a listbox, but also a value property. The value property might be identical to the text property, or it might contain a completely different piece of data, such as an identification number. For example, you might choose to display product names in a listbox or a drop down listbox, but set the values for each displayed item to their respective product IDs. When a user selects a product name, the product ID is passed to the application, rather than the name of the product itself. One benefit of this approach is to eliminate processing confusion between products with similar names.

Using DB Web Controls in Master-Detail Applications

DB Web Controls allow you to build full-fledged master-detail applications, using the DBWebDataSource, DBWebGrid, and DBWebNavigator controls. To support master-detail applications, these controls must provide a way to specify cascading behavior.

This topic includes information about:

- Specifying Cascading Deletes
- Specifying Cascading Updates

Cascading Deletes

In a master-detail application, the application typically uses an OnApplyChanges event to send the DataSet changes to the server. It is necessary for the master data adapter's update method (in BDP.NET, the AutoUpdate event) to be called prior to the detail data adapter's update method. Otherwise, insertion of detail rows fails if the master row has not yet been inserted. If the master row is deleted prior to the detail row, the server might return an error.

The property CascadingDeletes has been added to the DBWebDataSource control. The CascadingDeletes property specifies how the server deletes rows in master-detail applications. The CascadingDeletes property provides the following three options:

- NoMasterDelete (Default)
- ServerCascadeDelete
- ServerNoForeignKey

Note: When DB Web Controls are connected to a DataTable that is a detail table in a relation, the control's rows are automatically limited to the rows controlled by the current parent row in the master table.

NoMasterDelete

This option does not allow deletion of a master row containing detail rows. This option should be used when the server enforces a foreign constraint between master and detail, but it does handle cascading deletes. You must:

1. Delete detail rows.
2. Apply the changes with an apply event (for example, the BdpDataAdapter. AutoUpdate event).
3. Delete the master row.
4. Call the apply event (for example, the BdpDataAdapter. AutoUpdate event).

This option is the default value for the CascadingDeletes property.

ServerCascadeDelete

This option allows deletion of the master row. This option should be specified whenever the server is set up to automatically handle cascading deletes. When a master row is deleted, the detail rows will automatically disappear from view. Any time prior to applying the change, you can undo the parent row deletion and all the detail rows come back into view. If the server is not set up to handle cascading deletes, an error may occur when attempting to send changes to the server.

ServerNoForeignKey

This option automatically deletes all detail rows whenever a master row is deleted. This option should be specified whenever there are no foreign key constraints between the master-detail tables on the server. Like the

ServerCascadeDelete option, when a master row is deleted, the detail rows will automatically disappear from view. Any time prior to applying the change, it is possible to undo the master row deletion to redisplay the detail rows. If you specify this option and foreign key constraints exist between master and detail tables, an error will be thrown by the server when attempting to delete the master table.

Cascading Updates

In a master-detail application, the application typically uses an OnApplyChanges event to send the DataSet changes to the server. It is necessary for the update method of the master data adapter (in BDP.NET, the AutoUpdate event) to be called prior to the update method of the detail data adapter. Otherwise, insertion of detail rows fails if the master row has not yet been inserted. If the master row is deleted prior to the detail row, the server might return an error.

The property CascadingUpdates has been added to the DBWebDataSource control. This property specifies how the server updates foreign-key values in master-detail applications. The CascadingUpdates property provides the following three options:

- NoMasterUpdate (default)
- ServerCascadeUpdate
- ServerNoForeignKey

Note: When DB Web Controls are connected to a DataTable that is a detail table in a relation, the rows of the control are automatically limited to the rows controlled by the current parent row in the master table.

NoMasterUpdate

This option does not allow changes to the foreign key value of a master row if it has any associated detail rows. This option is the default value for the CascadingUpdates property.

ServerCascadeUpdate

This option allows you to change the foreign key value of the master row. You should use this option whenever the server automatically handles cascading updates. When the foreign key value of a master row is changed, the key value is changed automatically in the detail rows. Anytime prior to applying the change, you can undo the change to the master row and all the detail key changes will be undone also. If the server is not set up to handle cascading updates, an error might occur when attempting to update the changes to the server.

ServerNoForeignKey

This option also allows changing the foreign key value of the parent row, but should be used whenever there is no foreign key between the master and detail tables on the server.

Using XML Files with DB Web Controls

The DBWebDataSource component provides a way for you to create and use XML and XSD files as the data source for an ASP.NET application. Typically, you only use these types of files with the DBWeb controls as a way of prototyping your application. By using XML files as the data source, you can eliminate potentially costly database resources during the design and development phase of your project.

This topic covers the following issues.

- XML files as data sources.
- Suggested workflow strategy.
- Authentication and caching issues.

XML Files as Data Sources

XML has become another standard data source for many applications, but for ASP.NET applications in particular. When working with data that does not require strong security and therefore can be sent over HTTP as text, XML files provide a simple solution. Because the files are text, they are easy to read. Because the XML tags describe the data, you can understand and process the data structures with little difficulty.

Despite their obvious advantages over more complex data structures, XML files do have some drawbacks. For one thing, they are not secure, therefore, it is not a good idea to pass sensitive data, such as credit card numbers or personal identification (PIN) numbers, over the Internet by way of XML files. Another drawback is the lack of concurrency control over XML records, unlike database records.

Nonetheless, the self-describing nature and the lightweight data format of XML files makes them a natural choice as data sources for ASP.NET applications. The DBWebDataSource control, in particular, has been built to handle XML files as well as other types of data sources. There are no special requirements for using XML files, no unique drivers or communication layers beyond those that come with Developer Studio 2006, so you will find it easy to work with XML files as data sources.

Suggested Workflow Strategy

You use the DBWebDataSource control to create the XML file for your application and to connect the XML file with a DataSet object. The basic workflow strategy is this:

- Build an ASP.NET application, with a connection to your target database. Use DBWeb controls, including a DBWebDataSource and specify a non-existent XML file. When you run the application, your DataSet receives the result set from the target database and the DBWebDataSource then fills the XML file with tagged data representing the DataSet.
- From this point forward, you can eliminate the data adapter and data connection, keeping only a DataSet, the DBWebDataSource, and the reference to the XML file. Your DBWeb controls will pull data from the XML file and DataSet rather than from the database. For more information, follow the links to specific procedures on building and using XML files with DBWeb controls.

Authentication and Caching Issues

The DB Web Controls support automatic reading of an XML file by the DBWebDataSource component at both designtime and runtime. To support XML files, the DBWebDataSource component includes caching properties. If you use XML caching, the XML file data is automatically read into the DataSet whenever a data source is loaded.

If you do not implement user authentication in your application, you will likely only use this feature for prototyping. Otherwise, without user authentication, users may experience permissions errors when trying to access a single

XML file concurrently. When multiple clients are using the application, the XML file is constantly being overwritten by different users. One way to avoid this is to write logic in your server application to check row updates and notify various clients when there is a conflict. This is similar to what a database system does when it enforces table-level or row-level locking. When using a text file, like an XML file, this level of control is more difficult to implement.

However, if you implement user authentication, you can create a real-world application by setting the UseUniqueFileName property. This property specifies that the DBWebDataSource control will create uniquely named XML files for each client that uses accesses the XML file specified in the XMLFileName property of the DBWebDataSource. This helps avoid data collisions within a multi-user application. The drawback to this approach is that each XML file will contain different data and your server application will need built-in logic to merge the unique data from each client XML file.

Read-write applications using XMLFileName require that all web clients have write access to the XML files to which they are writing. If the web client does not have write access, the client will get a permissions error on any attempt to update the XML file. You must grant write access to the clients who will use the application.

DB Web Control Wizard Overview

The Borland DB Web Controls are data-aware web components. These DB Web Controls allow you to encapsulate data-aware functionality into standard web controls. One benefit of this approach is that the data binding function is fulfilled by the control itself, eliminating the need to add a call to the DataBind method.

The basic concepts involved in creating DB Web Controls are:

- The ASP.NET Control Execution Lifecycle
- Data Binding Concepts
- Overriding ASP.NET Methods
- Implementing DB Web Interfaces
- Essential Code Modifications

The ASP.NET Control Execution Lifecycle (CEL)

Anytime an ASP.NET web forms page is displayed, ASP.NET performs what Microsoft calls the CEL. This consists of a number of steps, which are represented by methods:

Initialize
Load view state
Process postback data
Load
Send postback change notifications
Handle postback events
Prerender
Save state
Render
Dispose
Unload

You can add logic to any or all of these events by adding code to given methods, such as the Page_Load method or the OnInit method. Most often, however, you will need to override the Render method.

Data Binding

In ASP.NET you can bind to a variety of data sources including databases, text files, XML files, arrays, and collections. In Developer Studio 2006, controls provide a simple property-based interface to data sources. In the **Object Inspector**, you can bind a selected control to a data source that is identified to your project by way of the BDP.NET controls, SQL client controls, or other data or file controls. Each type of data control has different sets of binding requirements. For instance, any collection control, such as the listbox control, data grid, or listview control, must bind to a data source that implements the ICollection interface. Other controls, like buttons and text boxes, do not have this requirement.

When you are programming with web controls, you must add the code to perform the data binding. For example, if you created an instance of a data grid, the command that you would add would look like:

```
dataGrid1.DataBind();
```

When using DB Web Controls, you no longer need to add this code. DB Web Controls handle the data binding operation for you. The DBWebDataSource component serves as a bridge between your data source component

and the specific DB Web control you want to use. The DBWebDataSource creates and manages the data binding between the data source and the control. Although you can physically add the code to instantiate a DB Web control and to perform the data binding, it is unnecessary to do so. You can drop your components onto a web form and select the linkages from property drop down list boxes in the **Object Inspector**.

Note: When creating a new DB Web control or extending an existing control, you may need to add code to perform binding of some properties.

Overriding ASP.NET Methods

The main method you will need to override is the Render method (or the RenderContents method). The Render method is responsible for displaying your controls visibly on the web page. When you define Rendere the Render method and pass it an instance of the HtmlTextWriter class, you are indicating that whatever you code in the method is to be written to the ASP.NET page in HTML. The Write method of the HtmlTextWriter class writes a sequential string of HTML characters onto a Web Forms page.

The following example shows how the control is declared in the file that is built by the **DB Web Control Wizard**. This is only a small segment of the code that is provided for you.

```
/// TWebControl1 inherits from the WebControl class of System.Web.UI.WebControls.

TWebControl1 = class(System.Web.UI.WebControls.WebControl)
```

When creating your own controls or extending existing controls, you must override the Render method to display your control. The Render method is responsible for sending output to an instance of an HtmlTextWriter class. HtmlTextWriter sends a sequence of HTML characters to the web forms page. The HTML characters are the representation in HTML of your control. For example, a web grid control is represented on a web forms page as an HTML table. Each control has its own HTML representation, but when you extend a control, you need to modify how the HTML is emitted to accurately represent your new control.

```
/// The following lines declare the Render method.
/// Output represents an instance of the HtmlTextWriter class.
/// HtmlTextWriter is the class that writes HTML characters to
/// the ASP.NET Web Forms page.

 strict protected
    procedure Render(Output: HtmlTextWriter); override;

implementation

{$REGION 'Control.Render override'}

/// The following procedure is the overridden Render method
/// You can include additional logic in the procedure to affect
/// the behavior of the control. This method, as written, does
/// nothing but write out a sequence of HTML characters that
/// define TWebControl1.

procedure TWebControl1.Render(Output: HtmlTextWriter);
begin
  Output.Write(Text);
end;
```

You would need to implement the preceding code even if you were trying to extend the capabilities of a standard web control. To extend one of the DB Web Controls you need to make more adjustments to this code.

Implementing DB Web Interfaces

When you run the **DB Web Control Wizard**, the wizard creates a code file for you, containing the basic code you need to extend a DB Web control. This file is similar to the file you would create if you were trying to extend a standard web control. The major difference is that the **DB Web Control Wizard** adds implementations of specific DB Web interfaces, which provide automatic access to a data source, tables, columns and their respective properties. Because the DB Web Controls handle so much of the postback and data binding automatically, you need to implement several specific interfaces to add this functionality to your control.

Essential Code Modifications

When you create a new DB Web Control Library, the **DB Web Control Wizard** creates a file template for you. This file contains the major elements you need to include in your project to create or extend a control. You will need to add or modify the following elements:

- Change the ToolboxBitmap attribute to specify your own icon for the Tool Palette, if necessary.
- Change the control declaration to specify the control you intend to inherit.
- Declare the correct Render method.
- Implement the IDBWebDataLink interface.
- Implement the IDBWebColumnLink and IDBWebLookupColumnLink interfaces, if necessary.
- Modify or extend the Render method.
- Modify hidden field registration, if necessary.
- Set data binding on specific properties, if necessary.

Change the ToolboxBitmap Attribute

If you have a bitmap icon available for use in the Tool Palette, specify its path in the ToolboxBitmap attribute in the DB Web Control Library file. The code might look something like this:

```
[ToolboxBitmap(typeof(WebControl1)]
['WebControl1.bmp')]
```

Make sure that you include the bitmap file in your project.

Change the Control Declaration

You can specify the ancestor more specifically. For example, if your control is an extended version of a DBWebGrid control, the code would look like this:

```
    MyDataGrid = class(Borland.Data.Web.DBWebGrid, IPostBackDataHandler, IDBWebDataLink)
```

Declare the Correct Render Method

Your control can inherit from either the Control namespace or the WebControls namespace. WebControls actually derives from the Control namespace.

The major difference for you is that WebControls defines all of the standard web controls, so if you plan on extending the capabilities of a web control like a textbox or a data grid, your control needs to inherit from WebControls.

By inheriting from WebControls, you are able to use all of the appearance properties of your base control. Typically, if you want to create a control that has a UI, inherit from System.Web.UI.WebControls. In the DB Web Control Library file, you will override the RenderContents method.

If your control inherits from Control, you need to supply the UI definition when you override the Render method. Typically, if you want to create a control that has no UI, you inherit from System.Web.UI.Control. In the DB Web Control Library file, you will override the Render method.

Implement the IDBWebDataLink Interface

This interface provides the access to a data source. You need to implement this interface for any DB Web control you intend to extend. The implementation is handled for you in the DB Web Control Library file.

Modify or Extend the Render Method

In the Render or RenderContents method, depending on which namespace you inherit from, you can override the properties of the base class. In the DB Web Control Library file the following code is automatically included for you:

```
procedure TWebControl1.Render(Output: HtmlTextWriter);
    begin
      Output.Write(Text);
    end;
```

This method passes the definition of your control to an instance of HtmlTextWriter, called Output in this case. The Text property will contain the HTML text that is to be rendered. If you wanted to code directly within the method, You could add code, as follows:

```
procedure TWebControl1.Render(Output: HtmlTextWriter);
    begin
     Output.WriteFullBeginTag("html");
         Output.WriteLine();

         Output.WriteFullBeginTag("body");
         Output.WriteLine();

         Output.WriteEndTag("body");
         Output.WriteLine();

     Output.WriteEndTag("html");
         Output.WriteLine();
    end;
```

This results in an ASP.NET web page with the following HTML code:

```
<html>
    <body>
    </body>
</html>
```

The use of the Text property, however, makes the code easier to work with. Once you have defined your control and its properties, along with various HTML tags, you can pass the entire structure to the Text property. From that point

forward, you need only refer to the Text property to act upon the control. You define the properties of your control and pass them to the HtmlTextWriter by creating a Text property that contains the control definition. It is instructive to look at the source code for some of the existing DB Web Controls. For example, the following code shows the definition of the Text property for the DBWebNavigator control.

```
protected string Text{
  get
       {

// Create a new instance of StringWriter.
                    StringWriter sw = new StringWriter();

// Create a new instance of HtmlTextWriter.
                    HtmlTextWriter tw = new HtmlTextWriter(sw);

// Call the DataBind procedure.
                    DataBind();

// Call the AddButtons procedure.
     AddButtons();

// Call the SetButtonsWidth procedure.
     SetButtonsWidth();

// Add a style to a panel.
     ClassUtils.AddStyleToWebControl(FPanel, this.Style);

// Render the HTML start tag for a panel control.
        FPanel.RenderBeginTag(tw);

// Call the HtmlTextWriter.Write method and pass the table
//    and tablerow tags to the web forms page.
        tw.Write("<table><tr>");


// If the ButtonType is set to ButtonIcons, iteratively create and render buttons
// to the web forms page.

     if( ButtonType == NavigatorButtonType.ButtonIcons )
   {
                    for( int i = 0; i < IconNavButtons.Count; i++ )
                    {

// Write the first table cell tag.
         tw.Write("<td>");

// Instantiate an image button.
               ImageButton b = (IconNavButtons[i] as ImageButton);

// Render the button on the web page.
                        b.RenderControl(tw);

// Write the closing table cell tag.
      tw.Write("</td>");
                    }
   }
  else

        // If the ButtonType is something other than ButtonIcons, iteratively create and
  // Render default navigation buttons to the web forms page.
```

```
      {
    for( int i = 0; i < NavButtons.Count; i++ )
          {

// Write the first table cell tag.
        tw.Write("<td>");

// Instantiate a button.
              Button b = (NavButtons[i] as Button);

// Render the button on the web page.
              b.RenderControl(tw);

// Write the closing table cell tag.
     tw.Write("</td>");
          }
   }

// Write the closing row and table tags.
 tw.Write("</tr></table>");

// Render the Panel end tag.
 FPanel.RenderEndTag(tw);
 return sw.ToString();
    }
}
```

Modify Hidden Field Registration

The DB Web Control Library file includes a call to register a hidden field, which identifies the key for a read-write control. If you are creating a read-only control, you can remove or comment out this call. The call is as shown in the following sample:

```
Page.RegisterHiddenField(DBWebDataSource.IdentPrefix +
DBWebConst.Splitter + IDataLink.TableName, self.ID);
```

Set Data Binding on Specific Properties

If you need other properties data bound, other than the Text property, you can add that data binding code in the same location where you find that the Text property is being bound. Typically, there is a call to DataBind in the PreRender method. The DataBind procedure itself is similar to the following sample, taken from the DBWebLabeledTextBox control source code. You can see in the following code that a number of properties are set after checking to see if the FColumnLink (from the IDBWebDataColumnLink interface) is bound to some data source.

```
public override void DataBind()
{
    try
  {
    FTextBox.ReadOnly = FReadOnly;
       FTextBox.ID = this.ID;
          base.DataBind();
       ClassUtils.SetBehaviorProperties(FPanel, this);
    ClassUtils.SetOuterAppearanceProperties(FPanel, this);
    ClassUtils.SetSizeProperties(FPanel, this);
         if( !ClassUtils.IsEmpty(FLabel.Text) )
```

```
              {
                  ClassUtils.SetInnerAppearanceProperties(FLabel, this);
               SetProportionalSize();
           SetLabelFont();
           FTextBox.Text = null;
             }

  // If there is a data source.
            if( IColumnLink.DBDataSource != null )
                 {

// And if there is bound data.
                    if( FColumnLink.IsDataBound )
                    {

// Then set behavior properties.
                         ClassUtils.SetBehaviorProperties(FTextBox, this);

// Set appearance properties.
                         ClassUtils.SetAppearanceProperties(FTextBox, this);

 // Set size properties.
                         ClassUtils.SetSizeProperties(FTextBox, this);
                       object o = IColumnLink.DBDataSource.GetColumnValue(Page, IColumnLink.
TableName, IColumnLink.ColumnName);

// If the page and the table and column names are not null,
// it means there is already bound data.
//    Put the string representation of the page, table, and
// column names into the textbox.
                   if( o != null )

                           FTextBox.Text = Convert.ToString(o);


                   else

// Otherwise, clear the textbox and bind it and
// its properties to the specified
column.
                           FTextBox.Text = "";
                       FTextBox.DataBind();
    }
}
```

Deploying ASP.NET Applications

This topic provides information about:

- Web Server Requirements
- Pre-Deploy Recommendations
- The Developer Studio 2006 ASP.NET Deployment Manager

For additional deployment information, see the deploy.htm file located, by default, at C:\Program Files\Borland\BDS\4.0.

Web Server Requirements

Before deploying your application to a web server, consider the following web server requirements:

- Internet Information Services (IIS) 6.0 must be installed and operational on the web server.
- The .NET Framework must be installed on the web server.
- ASP.NET must be enabled on the web server.
- The ASPNET account on the web server must be configured with the correct permissions.

For information on installing IIS, see the documentation that accompanies your Windows operating system. For information on performing the other tasks listed above, see the link to ASP.NET platform requirements at the end of this topic.

Pre-Deploy Recommendations

Before you deploy your application, you should disable debugging and rebuild the application to make it smaller and more efficient:

- For a Delphi ASP.NET or C# application, update the application web.config file to disable debugging. For details, see the link to using the Deployment Manager at the end of this topic.
- For a C# application, choose **Project** ▸ **Options** and change the **Debug/Release** option set to the **Release** option set and recompile the application.

The Developer Studio 2006 ASP.NET Deployment Manager

While you can use the XCOPY command-line tool to copy your entire project directory to a web server, only a subset of those files are actually required for deployment. For example, the .aspx, .config, and .dll files are required, but the Delphi-specific files such as the .bdsproj, .dcuil, and .pas files are not required.

Developer Studio 2006 includes the ASP.NET Deployment Manager to assist you in deploying ASP.NET applications. You can use it to deploy to a remote computer by using a share or an FTP connection, or to your local computer.

When you add a Deployment Manager to your project, an XML file (.bdsdeploy) is added to the project directory and a **Deploy** tab is added to the IDE. You provide destination and connection information on the **Deploy** tab and optionally modify the suggested list of files to copy, then the Deployment Manager copies the files to the deployment destination.

Building Web Services with ASP.NET

Web Services is a programmable entity that provides a particular element of functionality, such as application logic. Web Services is accessible to any number of potentially disparate systems through the use of Internet standards, such as XML and HTTP. Applications built with ASP.NET Web Services can be either stand-alone applications or subcomponents of a larger web application and can provide application components to any number of distributed systems using XML-based messaging. Developer Studio 2006 provides a number of methods that can help you build, deploy, and use applications with ASP.NET Web Services. For more general information about Web Services, refer to the Microsoft .NET SDK Documentation.

In This Section

ASP.NET Web Services Overview
Introduces the ASP.NET Web Services architecture for providing application logic.

Web Services Protocol Stack
Describes the infrastructure that makes Web Services work.

ASP.NET Web Services Support
Describes ASP.NET Web Services application support.

Building an ASP.NET "Hello World" Web Services Application
Describes the minimum steps to build a web service to expose functionality over the Web.

Accessing an ASP.NET "Hello World" Web Services Application
Describes the minimum steps to create a client application to access a web service.

ASP.NET Web Services Overview

Web Services is an Internet-based integration methodology that enables applications, independent of any platform or language, to connect and exchange information. Web Services is tightly integrated with the ASP.NET model used for the .NET Framework. Unlike traditional native Windows applications, ASP.NET Web Services applications contain objects and methods that are exposed over the Web using simple messaging protocol stacks. Any client can invoke a Web Services application over HTTP using a WebMethod. Like any method that can be accessed by way of a simple Windows Form application, a WebMethod provides some defined functionality. Unlike other types of methods, however, the WebMethod is accessed by way of a web browser. For more general information about Web Services, refer to the Microsoft .NET Framework SDK Documentation.

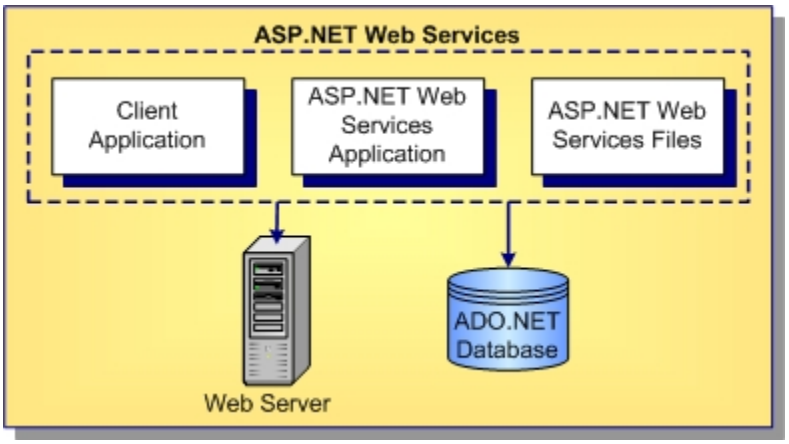
Borland provides tools to develop and access ASP.NET Web Services using a variety of techniques. As modular objects, web services can be reused without additional coding.

The following topics provide a brief introduction to the architecture of ASP.NET Web Services, the basic fundamentals of Web Services communication, and to the files created when you develop ASP.NET Web Services.

This topic introduces:

- ASP.NET Web Services Architecture
- Web Services Prerequisites
- Web Services Scenarios
- ASP.NET Web Services Files

ASP.NET Web Services Architecture



The major components of the ASP.NET Web Services architecture include a client application, an ASP.NET Web Services application, several files such as code files in the development language, .asmx files, and compiled .dll files. You need a web server to house both ASP.NET Web Services application and the client. Optionally, you might include a database server for storage and access of ASP.NET Web Services data.

Web Service Prerequisites

Before you begin developing a Web Services application, become familiar with the following concepts:

- **XML (Extensible Markup Language).** XML is a user-defined, human-readable structural description of data. Any data, dataset, or document that you intend to send to, or receive from, a web service is formatted in XML.
- **SOAP (Simple Object Access Protocol).** SOAP is the standard messaging protocol that is used for communication between web services and their clients. SOAP uses XML to format its messages, and contains the parameters or return values needed by servers and clients.

- **WSDL (Web Services Description Language).** WSDL is the language that describes a web service. A web service can be defined in any number of implementation languages. As a single-purpose utility, each web service must publish a description of its interface, which allows clients to interact with it. The WSDL document, at a minimum, describes the required parameters a client must provide and the result a client can expect to receive. The result description typically consists of the return data type.
- **UDDI (Universal Description, Discovery, and Integration).** UDDI is an industry initiative that provides a standard repository where businesses can publish web services for use by other companies. The UDDI repository contains links to, and descriptions of, a variety of web services. You can use the UDDI browser in the IDE to locate web services, download WSDL documents, and access additional information about web services and the companies that provide them.

Web Service Scenarios

Current web services provide simple information sources that you can easily incorporate into applications, such as stock quotes, weather forecasts, and sports scores. As the demand for access to business logic over the web increases, companies are finding ways of providing their customers with a class of applications to analyze and aggregate information. For example, a financial institution might provide a web service to consolidate and continuously update customer financial information, such as stock portfolio, 401(k), bank account, and loan information for display in a spreadsheet, web site, or a personal digital assistant (PDA). This saves customer from having to manually collect and combine the information on their own. Although much of this information is available through the web today, a web service will simplify accessing and consolidating information and will ensure greater reliability.

You can use web services for solutions in the following areas:

- **Enterprise Application Integration (EAI).** A web service could allow multiple business partners to exchange inventory, order, or financial data, for example, without specifically knowing the precise data layout in which data is stored for each partner. For instance, many customer relationship management (CRM) or other front-end applications store customer data in a format that is not entirely compatible with the way a back-end enterprise resource planning (ERP) system stores its financial or inventory information. However, a sales organization may wish to use its CRM solution to process real-time orders with up-to-date inventory information from the ERP system. A web service could be a solution to managing the transformation of CRM requests to ERP storage and from ERP responses to CRM confirmations.
- **Business-to-business (B2B) integration.** Similar to the EAI solution, a B2B solution could take advantage of a Web Services capability to provide cached data for large orders. B2B transactions, unlike business-to-consumer (B2C) transactions, often consist of high-volume transactions that would be prohibitive to execute at the level of a B2C transaction. For instance, a consumer might order one box of pencils from an online stationery store, but a business might order a thousand boxes monthly, with multiple shipping addresses. The scale and complexity of a B2B transaction requires the intervention of a web service to help simplify and process the transaction quickly and with consistency.
- **Business-to-consumer integration.** B2C web services typically manage web-based transactions. For example, a web service that allows you to look up postal codes eliminates the need for businesses to create a new program every time the service is included on a web site. Some commerce sites might use web services to help manage currency conversion when taking international sales orders.
- **Mobile (Smart client applications).** Because the small footprint of a mobile client requires that memory usage be reserved for only the most important system functions, and because mobile clients are, by definition, linked to the Internet by way of their wireless communication protocols, Web services play a vital role in providing lightweight but powerful applications to mobile devices. Web services allow mobile device users to perform a variety of tasks which require little more than data input at the device and data display of the results. All processing can occur on a remote web service, thus decreasing bandwidth requirements on the mobile device itself.
- **Distributed and Peer-to-Peer.** For certain types of distributed and peer-to-peer applications, web services play an important role. If you use distributed computing over an uncontrolled network (such as the Internet) rather

than over a LAN or corporate network, you might use web services. Web services do not require state maintenance, thus offering potentially improved performance, particularly where a request-response behavior is not absolutely required. For applications that require strict request-response behavior and high security, you should consider using an older, more controlled model, such as COM, CORBA, or .NET remoting.

ASP.NET Web Services Files

Certain files are automatically generated when you create applications with ASP.NET Web Services. These files enable the ASP.NET Web Services to render their services through a web server. The following table lists the files and their descriptions.

File	Description
.asmx	When you create an ASP.NET Web Services application, a text file is automatically generated with the .asmx extension. The required Web Services directive is placed at the top of this file to correlate between the URL address of the web service and its implementation. Within the .asmx file, you add Web Services logic to the methods visible by the client application. The .asmx file acts as the base URL for clients calling the XML web service. This file is compiled into an assembly, along with other files, for deployment.
code-behind	When you create an ASP.NET Web Service application, a code-behind file is generated with a language-specific extension. You add your Web Services logic to the public method to process Web Services requests and responses.
compiled DLL files	Web Services DLL files provide dynamic services on the web server.
.wsdl	This file is generated when you click the **Add Web Reference** feature to add the web service to your client application. It describes the Web Services interface available to the client.
.map	This file enables the discovery of a web service that is exposed on a given server. It also contains links to other resources that describe the web service.

Web Services Protocol Stack

Understanding the Web Services infrastructure requires that you have some exposure to Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). Because the infrastructure already exists, as a developer of XML web services, you can leverage the existing technology by using standard Web protocols such as XML and HTTP.

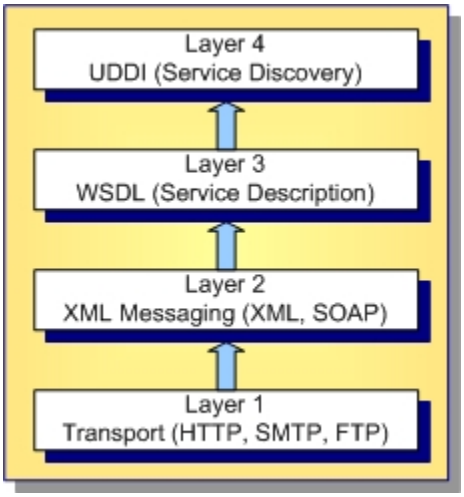
Borland provides an easy way to create, deploy, and use web services without concern for back-end processing so you can focus more on designing your services.

This topic provides the conceptual background to understand how the protocol stack contributes to Web Services functionality:

- How web services access and expose their services via the Web
- How XML passes information through standard SOAP and HTTP
- How a client can identify a web service offering
- How web services are discovered and accessed

Layers of the Web Services Protocol Stack

Web services consist of sets of internet protocols and standards for exchanging data between applications. The Web Services Protocol Stack describes the layering of the set of internet protocols or rules used to design, discover, and implement web services.



The major components or layers of a Web Service Protocol Stack include:

- **Transport Layer**—transports messages between applications
- **XML Messaging Layer**—encodes messages in XML that can be understood by both client and server
- **WSDL Layer**—describes the service provided
- **UDDI Layer**—centralizes services with a common registry

Transport Layer

The Transport layer is the first component in the stack and is responsible for moving XML messages between applications. The Transport protocol most commonly used is the standard HTTP protocol. Other commonly used Web protocols are SMTP and FTP.

XML Messaging

The messaging layer in the protocol stack is based on an XML model. XML is widely used in Web Services applications and is the foundation for all web services. XML is just one of the standards enabling web services to map between technology domains. You will find many resources on the Web that describe XML messaging. For more information, refer to the World Wide Web Consortium (W3C) site on Messaging listed in the link list below.

The XML Messaging specification is a broadly-defined umbrella under which a number of more specific protocols are defined. SOAP is one of the more popular standards, and is one of the most significant standards in communicating web services over the network. XML provides a means for communicating over the Web using an XML document that both requests and responds to information between two disparate systems. SOAP allows the sender and the receiver of XML documents to support a common data transfer protocol for effective networked communication. You will find many resources on the Web that describe SOAP. For more information, refer to the W3C site for SOAP listed in the link list below.

WSDL Layer

This layer represents a way of specifying a public interface for a web service. It contains information on available functions, on data types for XML messaging, binding information about the transport protocol, and the location of the specific web service.

Any client application that wants to know about a service, what data it expects to receive, whether or not it delivers any results, and the supported transport, uses WSDL to find that information. When you create a Web Service, it must be described and advertised to its potential customers before it can be used. WSDL provides a common format for describing and publishing that web service information. Typically, WSDL is used with SOAP, and the WSDL specification includes a SOAP binding.

Use Borland's **Add Web Reference** feature to obtain a WSDL document for your web service. The WSDL document, or proxy file, is copied to the client and is used to call the server. This proxy file is named References.*, where the file name extension reflects the language type. For more information about WSDL, refer to the W3C WSDL site listed in the link list below.

UDDI Layer

This layer represents a way to publish and find web services over the Web. You can think of this layer as the White and Yellow Pages of your phonebook. The White pages of web services provides general information about a specific company, for instance, their business name, description, and address. The Yellow Pages includes the classification of data for the services offered, for instance, industry type and products.

The protocol you use to publish your web services is known as UDDI. The UDDI Business Registry allows anyone to search existing UDDI data and enables you to register your company and its services. With Developer Studio 2006, your data automatically gets published to the registry, or a distributed directory for business and web services.

ASP.NET Web Services Support

ASP.NET Web Services support VCL.NET Forms, .NET Windows Forms, and ASP.NET Web Forms. These forms can be used to create client applications that access Web Services applications. Use the **Add Web Reference** feature to add the desired ASP.NET Web Services application to the client application. Using the **UDDI Browser** you can locate Web Services applications you might want to use.

Developer Studio 2006 provides simple tools to develop and deploy your ASP.NET Web Services applications. Additionally, Developer Studio 2006 helps you import WSDL documents that describe particular Web Services applications and expose their functionality to the client application. You can use the sample WebMethod provided by Developer Studio 2006, which lets you create and access an ASP.NET Web Services application.

This topic includes:

- ASP.NET Web Services Client Support
- ASP.NET Web Services Server Support
- ASP.NET Web Services Namespaces

ASP.NET Web Services Client Support

You can create a Web Services application that is simply a provider, or a server application. This application resides on a web server and can be accessed by any client that understands the application architecture. If you want to consume a Web Services application yourself, you need to create a client application. Developer Studio 2006 provides different tools you can use to build client applications:

- Windows Forms
- Web Forms
- Web References

Windows Forms Versus ASP.NET Web Forms

To determine the best type of form to use for your client application—Windows form or ASP.NET Web form—consider the type of service you want to access. In most cases, the service you choose will dictate which type of application you should create.

If you need to provide a rich application that can process complex content on a client workstation, or that can use a web service application as a supporting piece for a rich client application over a secure network connection, you might consider building a Windows Forms application. If you need to provide a thin-client application that performs simple data manipulation or satisfies a single-purpose requirement, consider using ASP.NET Web Forms. Web Forms are platform-independent interfaces that display in a web browser and invoke Web Services applications over a simple protocol like HTTP.

You can also create an ASP.NET Web Services application as a console application which can be accessed through either a console window, or by another Web Services application, even one without a client.

Add Web Reference

You can add a Web Reference to your client application to access web services. A Web Reference refers to either a WSDL document or an XML schema, which is imported into your client application. The WSDL document or XML schema describes a web service. When you import one of these documents, Developer Studio 2006 generates the interfaces and class definitions needed for calling that web service. Right-click the **WebService** node in the **Project Manager** and select **Add Web Reference**. A **UDDI Browser** appears. To add the web service to your client application, you must navigate within the browser and locate the WSDL document for the web service.

ASP.NET Web Services Server Support

The ASP.NET Web Services application you build in Developer Studio 2006, provides programmatic access to the application logic of one or more web services. You define the services you want to expose, how the services are to be used, and the infrastructure that receives and processes requests and responses.

When you create a new ASP.NET Web Service application, the **New ASP.NET Application** dialog box lets you specify the name and location of the ASP.NET Web Services application, and automatically creates the files required for deployment. When you specify the application settings, Developer Studio 2006 generates the .asmx file that acts as a base URL for clients calling the ASP.NET Web Services application.

ASP.NET Web Services Namespaces

For more information on System.Web.Services namespaces, refer to the Microsoft .NET Framework SDK.

Building Applications with Windows Forms

Windows Forms provide a traditional approach to developing user interfaces, client/server applications, forms, controls, and application logic. Windows Forms fully leverage the .NET Framework. This section provides an overview of Windows Forms using Developer Studio 2006 and common steps to building a simple Windows project.

In This Section

Windows Forms Overview
Introduces the Windows Forms architecture for building Windows applications using Developer Studio 2006.

Building a Windows Forms Application
Describes the essential tasks to create a Windows Forms application using Developer Studio 2006.

Deploying Windows Forms Applications
Provides general information about deploying Windows forms applications.

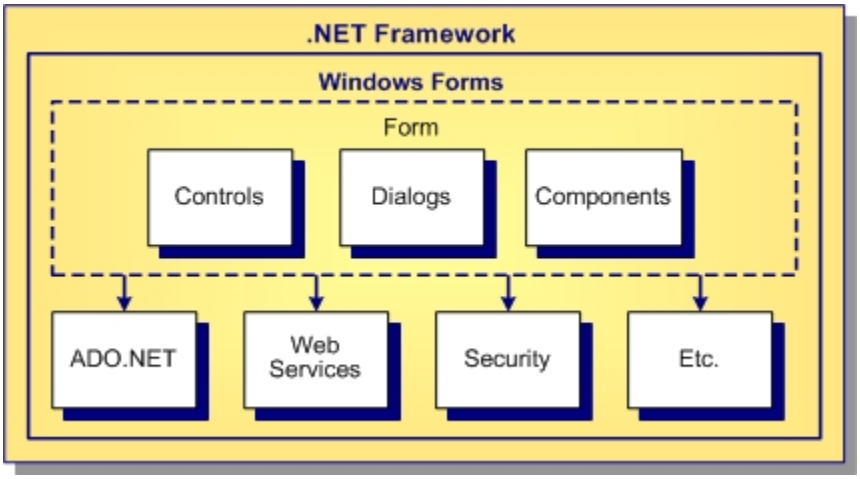
Windows Forms Overview

Windows Forms is the .NET programming environment for building native Windows applications in a managed environment. Building Windows clients with .NET allows applications to use features unavailable to browser clients while leveraging the .NET Framework for general infrastructure. Windows Forms combines features of both traditional and Internet-centric development, presenting a programming model that takes advantage of a unified .NET Framework (for instance, for security and dynamic application updates) and the richness of GUI Windows clients.

This section includes:

- Windows Forms Architecture
- Windows Forms Components
- Windows Forms Data Access
- Windows Forms Namespace

Windows Forms Architecture



Windows Forms share common .NET Framework with other programming models, like ASP.NET and ADO.NET.

Windows Forms

Developer Studio 2006 provides an IDE for creating GUI applications in a RAD environment. Developers drag controls, dialogs, and components onto the form **Designer**, set properties in **Object Inspector**, and code the logic to respond to events.

Windows Forms Components

The **Tool Palette** for Windows Forms in Developer Studio 2006 provides components, controls, and dialogs for designing a GUI. Components are classes that represent reusable objects. Controls are a type of component with user interface functionality. (All controls are components, but not all components are controls.) Typically, you design user interfaces by positioning and sizing components and controls on a form. Examples of common controls and components include buttons and menus. To facilitate the construction of menus, Developer Studio 2006 provides a menu designer for main menu and context menu components. Dialog boxes are a type of form, which in turn can contain controls. Dialogs provide for various types of user interaction.

Windows Forms Data Access

Within the .NET Framework, Windows Forms access data through ADO.NET. You can connect a Windows application to an ADO.NET data source using data components included in the .NET Framework and BDP.NET. BDP.NET components connect to a number of industry standard databases. For more information, see the ADO.NET section.

Windows Forms Namespace

Common Windows Forms classes like Form and Menu are contained within the System.Windows.Forms namespace. The namespace also contains controls like Button, CheckBox, and Label. Use the **Object Inspector** in Developer Studio 2006 to set properties, methods, and events within Windows Forms classes.

Deploying Windows Forms Applications

For the common language runtime, deploying Windows Forms applications requires installation of the .NET Framework on the target computer. If the Windows Forms application is simple, consisting of a single executable, the .exe file may reside unregistered in the appropriate program directory. If the Windows Forms application includes a shared assembly, the assembly must be installed to the Global Assembly Cache using tools in the .NET Framework. For more information, see the .NET Framework SDK help.

Building Applications with VCL.NET Components

VCL.NET is an extended set of the VCL components that provide a way to quickly build advanced applications in Delphi. With VCL.NET you can provide your Delphi VCL applications and components to Microsoft .NET Framework users. With Developer Studio 2006 you gain the benefit of the .NET Framework along with the ease-of-use and powerful component-driven application development of Delphi.

Developer Studio 2006 provides distinct application types for your use: you can create VCL.NET form applications that run on the .NET Framework that use VCL.NET components and controls; you can create .NET Windows Forms applications that use the underlying .NET Framework and .NET controls while offering Developer Studio 2006 code-behind; you can create powerful ASP.NET applications that use the underlying .NET Framework, ASP.NET controls, and also offer Developer Studio 2006 code-behind. The following topics provide more information on how to take advantage of the new VCL.NET provisions in Developer Studio 2006.

In This Section

VCL for .NET Overview
Introduces the VCL for .NET architecture for building applications using Developer Studio 2006.

Porting VCL Applications
Provides conceptual information about porting issues.

Building a VCL Forms Application
Describes the essential tasks to create a VCL Forms application using Developer Studio 2006.

VCL for .NET Overview

VCL for .NET is the programming framework for building Developer Studio 2006 applications using VCL components. Developer Studio 2006 and VCL for .NET are intended to help users leverage the power of Delphi when writing new applications, as well as for migrating existing Win32 applications to the .NET Framework.

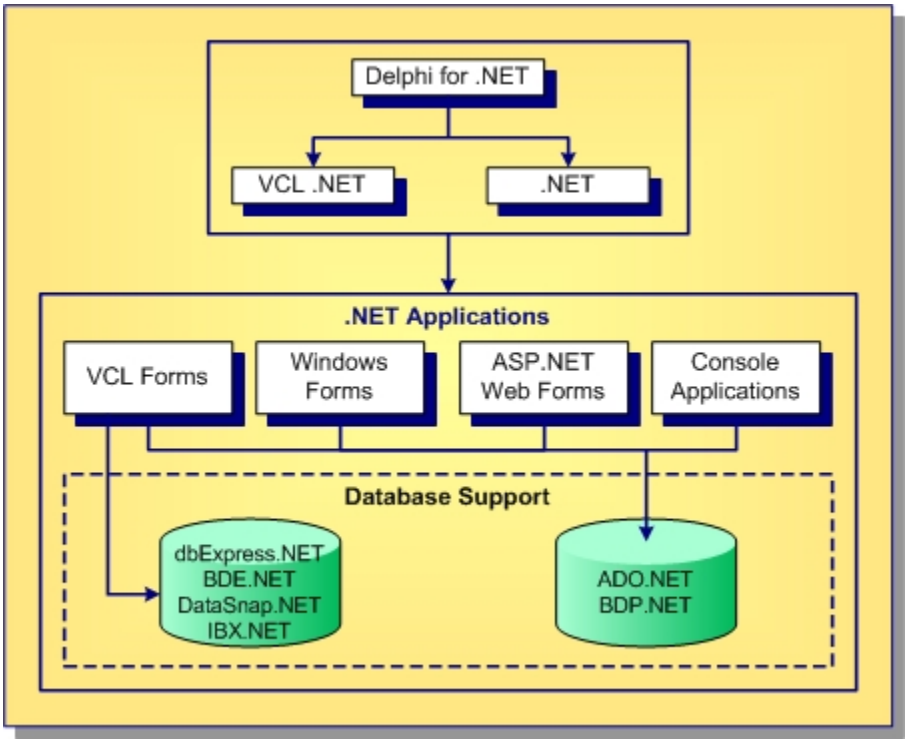
These technologies allow a Delphi developer to migrate to .NET, taking their Delphi skills and much of their current Delphi source code with them. Developer Studio 2006 supports Microsoft .NET Framework development with the Delphi language and both VCL for .NET controls and Windows Forms controls. Developer Studio 2006 ASP.NET also supports WebForms, and SOAP and XML Web Services application development.

VCL for .NET is a large subset of the most common classes in VCL for Win32. The .NET Framework was designed to accommodate any .NET-compliant language. In many cases Delphi source code that operates on Win32 VCL classes and functions recompiles with minimal changes on .NET. In some cases, the code recompiles with no changes at all. VCL for .NET is a large subset of VCL, therefore it supports many of the existing VCL classes. However, source code that calls directly to the Win32 API requires source code changes. Also, dependent third-party Win32 VCL controls need to be available in .NET versions for compatibility.

This section introduces:

- VCL for .NET Architecture
- VCL for .NET and the .NET Framework
- VCL for .NET Components
- Borland.VCL Namespace
- Porting Delphi Applications to Developer Studio 2006
- Importing .NET Components for Use in VCL for .NET Applications

VCL for .NET Architecture



VCL is a set of visual components for building Windows applications in the Delphi language. VCL for .NET is the same library of components updated for use in the .NET Framework. VCL for .NET and the .NET Framework coexist

within Developer Studio 2006. Both VCL for .NET and .NET provide components and functionality that allow you to build .NET applications:

- VCL for .NET provides the means to create VCL Forms applications, which are Delphi forms that are .NET-enabled, and use VCL for .NET components.
- VCL for .NET provides VCL non-visual components which have been .NET-enabled to access databases. You can also access databases through the ADO.NET and BDP.NET providers.
- .NET provides the means to build .NET Windows Forms, Web Forms, and Console applications, using .NET components, with Delphi code-behind.

You can build VCL Forms applications using VCL for .NET components, or Windows Forms applications using .NET components. You can also build ASP.NET Web Forms applications using either VCL for .NET components or .NET components.

VCL for .NET and the .NET Framework

The .NET Framework provides a library of components, classes, and low-level functionality that manages much of the common functionality, from the display of buttons to remoting functionality, without regard to the underlying implementation language. VCL for .NET and the .NET Framework are functionally equivalent. Like the .NET Framework, VCL for .NET provides libraries of components, controls, classes, and low-level functionality that help you build Windows Forms, Web Forms, and console applications that run on the current Windows .NET Framework platform.

VCL for .NET is not a replacement for the .NET Framework.

You will still need the .NET runtime to use VCL for .NET, but you can build complete applications using VCL for .NET components that will run on .NET platform.

You can build Developer Studio 2006 applications without using VCL for .NET, by creating Windows Forms, Web Forms, and Console applications using Developer Studio 2006 code.

You can use Developer Studio 2006 to create powerful .NET applications using .NET components, or VCL for .NET components that have been migrated from the Delphi VCL. If you have existing Delphi VCL applications that you want to run on Windows XP, you can easily port those applications by using Developer Studio 2006.

VCL for .NET Components

VCL for .NET consists of a set of visual and non-visual components. VCL for .NET builds on the concept of constructing applications visually, eliminating much manual coding.

Visual Components

Developer Studio 2006 provides a set of visual components, or controls, that you can use to build your applications. In addition to the common controls, such as buttons, text boxes, radio buttons, and check boxes, you can also find grid controls, scroll bars, spinners, calendar objects, a full-featured menu designer, and more. These controls are represented differently in Developer Studio 2006 than they are in frameworks, such as the .NET Framework.

In an IDE for other languages, such as C# or Java, you will see code-centric representations of forms and other visual components. These representations include physical definitions, such as size, height, and other properties, as well as constructors and destructors for the components. In the **Code Editor** of Developer Studio 2006 you will not see a code representation of your VCL for .NET components.

Developer Studio 2006 is a resource-centric system, which means that the primary code-behind representations are of event handlers that you fill in with your program logic. Visual components are declared and defined in text files with the extensions .dfm (Delphi Forms) or .nfm (Developer Studio 2006 forms). The nfm files are created by

Developer Studio 2006 as you design your VCL Forms on the Forms Designer, and are listed in the resource list in the Project Manager for the given project.

Non-Visual Components

You can use non-visual components to implement functionality that is not necessarily exposed visually. For example, you can access data by using non-visual components like the BDP.NET components, which provide database connectivity and DataSet access. Even though these components do not have visual runtime behavior, they are represented by components in the **Tool Palette** at designtime. VCL for .NET provides a variety of non-visual components for data access, server functions, and more.

Borland.VCL Namespace

VCL for .NET classes are found under the Borland.Vcl namespace. Database-related classes are in the Borland.Vcl.DB namespace. Runtime library classes are in the Borland.Vcl.Rtl namespace.

Unit files have been bundled in corresponding Borland.Vcl namespaces. In some cases, units have been moved. However, namespaces are identified in a way that will assist you in finding the functionality you want.

Source files for all of the Developer Studio 2006 objects are available in the c:\Program Files\Borland\BDS\4.0 \Source subdirectory.

Porting Delphi Applications to Developer Studio 2006

If you have existing applications written with an earlier version of Delphi, you might want to port them to .NET. In most cases, this will be easier than rewriting the applications. Because Developer Studio 2006 takes advantage of significant structural elements in the .NET Framework, you will need to perform some manual porting tasks to make your applications run. For example, the .NET Framework does not support pointers in safe code. So, any instance of a pChar or pointer variable will need to be changed to a .NET type. Many Delphi objects have been updated to accommodate these type restrictions, but your code may include references to pointers or unsupported types. For more information, refer to the Language Guide in this Help system.

Importing .NET Components for Use in VCL for .NET Applications

Developer Studio 2006 provides the **.NET Import Wizard** to help you import .NET controls into VCL for .NET units and packages. For example, you can wrap all .NET components, like those from the System.Windows.Forms assembly, in ActiveX wrappers that can be deployed on VCL for .NET applications. Once you have imported the .NET components of your choice, you can add a completed package file containing the units for each component to the **Tool Palette**. You can also view and modify the individual unit files, which can be useful reference material when you are writing your own custom components.

Porting VCL Applications

When porting VCL applications from Delphi 7 to Developer Studio 2006, there are issues you need to consider. Along with basic language elements that need to be replaced or modified, there are strategies that you should follow to make sure that you port your applications fully and reliably.

This topic includes

- General Language Issues
- Renaming Packages
- New Language Features
- Porting Web Service Client Applications

General Language Issues

Porting Delphi 7 applications to Developer Studio 2006 exposes several language issues in the .NET Framework. For instance, the .NET Framework considers pointers to be *unsafe* and so does not consider applications that use pointers to fall into the category of *managed* code. To be compliant with the .NET Framework, you need to modify your applications to avoid or circumvent the use of pointers, the pChar type, and other language-specific elements.

In addition, there are critical issues with the Win32 API, using crackers, migrating char types, and other topics.

Renaming Packages

When porting a Delphi 7 package to Developer Studio 2006, you will need to change the old package names in the "Requires" list to the corresponding new package names. The following table shows the old and new names.

Old Package Name	New Package Name
rtl	Borland.Delphi and Borland.VclRtl
vcl	Borland.Vcl
vclx	Borland.VclX
dbrtl	Borland.VclDbRtl
bdertl	Borland.VclBdeRtl
vcldb	Borland.VclDbCtrls
dbexpress	Borland.VclDbExpress
dbxcds	Borland.VclDbxCds
dsnap	Borland.VclDSnap
dsnapcon	Borland.VclDSnapCon
vclactnband	Borland.VclActnBand

Borland.VclActnBand Packages are now installed by using **Component** ▸ **Installed .NET Components** ▸ **.NET VCL Components**.

New Language Features

Several new features have been added to the Delphi language to support programming concepts and features of the .NET platform and the CLS:

- Partitioning code into namespaces

- New visibility specifiers for class members
- Class static methods, properties, and fields
- Class constructors
- Nested type declarations within classes
- Sealed classes
- Final virtual methods
- Operator overloads in classes
- .NET attributes
- Class helper syntax

Programming in the garbage-collected environment of .NET brings a number of new issues related to allocating and disposing of objects. These issues are discussed in Memory Management Issues on the .NET Platform.

Porting Web Service Client Applications

The .NET Framework employs a major architectural shift in how it handles web services and web service clients. Your existing web service client applications need to be modified to operate on the .NET Framework. Developer Studio 2006 does not support the RIO components, and uses a more transparent .NET approach to managing web service client applications. You will need to eliminate RIO components and modify the way you access WSDL documents.

Language Issues in Porting VCL Applications to Developer Studio 2006

The VCL in Developer Studio 2006 was created with backward compatibility as the primary goal. However, there are some ways in which the managed environment of .NET imposes differences in the way VCL applications must work. This document describes most of these differences, and indicates some of the steps you should take to port a VCL application to the .NET environment.

This document does not attempt to describe the new extensions to the Delphi language. It is limited to the way existing Delphi code maps to the new Developer Studio 2006 language and VCL framework. This document does contain links into specific topics within the Delphi Language Guide, where new language features are explained in detail.

This topic covers the following material:

- Migrating Pointer types
- Migrating Char and string types
- Creating and destroying objects
- Calling the Win32 API
- Migrating Variants
- Working with resources
- Change to `OnCompare`

Migrating Pointer Types

Pointer types are not CLS compliant, and are not considered "safe" in the context of the .NET Common Language Runtime environment. The port of the VCL has, therefore, eliminated pointers, replacing them with appropriate alternatives such as dynamic arrays, indexes into an array or string, class references, and so on. When porting a VCL application, one of the first steps is to locate where you use pointer types and replace them as appropriate.

Untyped Pointers

Untyped pointers are considered unsafe code. If your code includes untyped pointers, the .NET utility PEVerify will fail to verify it. Code that cannot be verified for type safety cannot be executed in a secured environment, such as a web server, SQL database server, web browser client, or a machine with restricted security policies.

In the VCL, untyped pointers have been replaced with more strongly-typed values. In most cases, where you used to find an untyped pointer, you will now find TObject. For example, the elements of TList are now of type TObject, rather than of type Pointer. Your code can cast any type to an object, and cast a TObject to any other type (even value types such as Integer, Double, and so on). Casting TObject to another type will generate a runtime error if the object is not, in fact, an instance of the type to which you are casting it. That is, this cast has the same semantics as using the as operator.

In some cases, the Pointer type has been replaced with a more precise type. For example, on TObject, the `ClassInfo` function returns a value of type Type rather than an untyped pointer.

Untyped pointers that were used for parameters whose type varied depending on context have typically been replaced by overloading the routine and using var parameters with the possible types. In the case of untyped pointers that are used with API calls to unmanaged code (such as the Windows API or calls to a data access layer such as the BDE) the untyped pointer is replaced with System.IntPtr. Thus, for example, the TBookmark type, defined in the `Db` unit, now maps to IntPtr.

Code that used the address operator (@) to convert a value to an untyped pointer must now change. When the untyped pointer has changed to TObject, usually all you need to do is eliminate the @ operator. On value types, you may need to replace the @ operator with a typecast to TObject, so that the value is "boxed". Thus, the following code

```
var
  P: Pointer;
  I: Integer;
begin
  I := 5;
  P := @I;
```

could be converted to

```
var
  P: TObject;
  I: Integer;
begin
  I := 5;
  P := TObject(I);
```

When the untyped pointer has changed to IntPtr, you need to use the Marshal class to allocate a chunk of unmanaged memory and copy a value to it, rather than just using the @ operator. Thus the following code:

```
var
  P: Pointer;
  R: TRect;
begin
  R := Rect(0, 0, 100, 100);
  P := @R;
  CallSomeAPI(P);
```

would be converted to

```
var
  P: IntPtr;
  R: TRect;
begin
  R := Rect(0, 0, 100, 100);
  P := Marshal.AllocHGlobal(Marshal.SizeOf(TypeOf(TRect)));
  try
    Marshal.StructureToPtr(TObject(R), P, False);
    CallSomeAPI(P);
  finally
    Marshal.FreeHGlobal(P);
  end;
```

Note: All unmanaged memory that you allocate using the Marshal class must be explicitly freed. The .NET garbage collector does not clean up unmanaged memory.

Procedure Pointers

A special case for untyped pointers is when they represent procedure pointers. In managed code, procedure pointers are replaced by .NET delegates, which are more strongly typed. Declarations of procedural types are delegate

declarations in Developer Studio 2006. You can obtain a delegate for a method or global routine using the @ operator. The code looks the same as obtaining a procedure pointer on the Win32 platform, so in many cases there is nothing you need to change when porting code. However, it is important to keep in mind that when you use the @ operator, you get a newly-created delegate, not a pointer.

If you are passing a procedure pointer to an unmanaged API using the @ operator, for example,

```
Handle := SetTimer(0, 0, 1, @TimerProc);
```

the only reference to the delegate is the one passed to the API call because the delegate is created on the fly. This means that the garbage collector will eventually dispose of the delegate after the return of the unmanaged API. If, as in this case, the unmanaged code may call the procedure after the return of the API call, you will encounter a runtime exception because the delegate no longer exists. You can work around this situation by assigning the delegate to a global variable, and passing the global variable to the unmanaged API.

When you call the Windows API `GetProcAddress` to obtain a procedure pointer, it is returned as an IntPtr. This value is not a delegate. You can't cast it to a delegate and call it. Instead, typically such code is translated to use Platform Invoke to call an unmanaged API. `GetProcAddress` is useful to determine whether the API is available so that you do not get a runtime exception when you use Platform Invoke. Thus, code such as the following:

```
type
  TAnimateWindowProc = function(hWnd: HWND; dwTime: DWORD; dwFlags: DWORD): BOOL; stdcall;
var
  AnimateWindowProc: TAnimateWindowProc = nil;
  UserHandle: HMODULE;
begin
  UserHandle := GetModuleHandle('USER32');
  if UserHandle <> 0 then
    @AnimateWindowProc := GetProcAddress(UserHandle, 'AnimateWindow');
  ...
  if AnimateWindowProc <> nil then
    AnimateWindowProc(Handle, 100, AW_BLEND or AW_SLIDE);
```

Would be translated to the .NET platform as follows

```
[DllImport('user32.dll', CharSet = CharSet.Ansi, SetLastError = True, EntryPoint =
'AnimateWindow')]
function AnimateWindow(hWnd: HWND; dwTime: DWORD; dwFlags: DWORD): BOOL; external;
var
  UserHandle: HMODULE;
  CanAnimate: Boolean;
begin
  UserHandle := GetModuleHandle('USER32');
  if UserHandle <> 0 then
    CanAnimate := GetProcAddress(UserHandle, 'AnimateWindow') <> nil
  else
    CanAnimate := False;
  ...
  if CanAnimate then
    AnimateWindow(Handle, 100, AW_BLEND or AW_SLIDE);
```

Note: The .NET example above is still late bound to the `AnimateWindow` API. An exception will not be generated when this code is loaded, if the DLL or function aren't available. The function call is resolved only when the code is executed for the first time.

String Pointers

Code that uses the PChar type usually serves one of three purposes:

- The type refers to a null-terminated string (especially when it is used with a Windows API call or an older RTL function).
- The type is used to navigate through a string when processing its value.
- The type is used to reference a block of bytes, relying on the fact that in Delphi for Win32, the Char type is a byte (the Char type is two bytes on the .NET platform).

In the first case, you can usually replace the PChar type with the type string. In the case of Windows API calls, the managed versions of the APIs now use a string or StringBuilder rather than a PChar, with the marshaling layer handling the conversions implicitly. Note that many of the RTL functions that supported the PChar type have been eliminated from the RTL, and you must replace them with corresponding versions that use the string type. The following table lists functions from the `SysUtils` units that have been eliminated because they relied on the PChar type, and the corresponding functions that use the string type:

PChar version	String version
AnsiExtractQuotedStr	AnsiDequotedStr or DequotedStr
AnsiLastChar, AnsiStrLastChar	(use index operator and string length)
AnsiStrComp, StrComp	CompareStr, AnsiCompareStr, WideCompareStr
AnsiStrIComp, StrIComp	CompareText, AnsiCompareText, WideCompareText
AnsiStrLComp, StrLComp	System.String.Compare (StartsStr)
AnsiStrLIComp, StrLIComp	System.String.Compare (StartsText)
AnsiStrLower, StrLower	AnsiLowerCase, WideLowerCase,
AnsiStrUpper, StrUpper	UpCase, AnsiUpperCase, WideUpperCase
AnsiStrPos, StrPos, AnsiStrScan, StrScan	Pos
AnsiStrRScan, StrRScan	LastDelimiter
StrLen	Length
StrEnd, StrECopy	(no equivalent)
StrMove, StrCopy, StrLCopy, StrPCopy, StrPLCopy	Copy
StrCat, StrLCat	+ operator, Concat
StrFmt	Format, FmtStr
StrLFmt	FormatBuf
FloatToText	FloatToStrF
FloatToTextFmt	FormatFloat
TextToFloat	FloatToStr

When a PChar type is used to navigate through a string, you must rewrite the code, replacing the PChar with an Integer that represents an index into the string. When rewriting such code, you must is recognize when you have reached the end of the string. When using the PChar type, there is a null character at the end of the string, and the code typically recognizes the end of the string by finding this null character. With a string-and-index approach, there is no such null character and you must use the string length to identify the end of the string. Be careful to check that the index is not past the end of the string before reading a character or you will get a runtime error.

Note: String data is immutable, so you can't write a single character into an existing string using a PChar. You can accomplish this using string indexing (e.g. `s[5]`), however.

When a PChar is used to reference a block of bytes, it is typically replaced by either an IntPtr or a dynamic array of bytes (`TBytes`). If replaced by an IntPtr, the issues in translating are the same as when replacing an untyped pointer. When replaced by `TBytes`, you my need to replace some PChar values with an index into the byte array if it is used to navigate the block of bytes. This is like replacing PChar with Integer to navigate through a string, except that indexes into `TBytes` are 0-based while indexes into strings are 1-based.

Writing Strings to Streams

In Delphi for Win32, it is common to find code similar to the following:

```
S1 := 'This is a test string';
Stream.WriteBuffer(S1[1], Length(S1));
```

On the Win32 platform, this code results in the entire string being written to the stream. On the .NET platform however, this same code produces a quite different result. On the .NET platform, the compiler generates a call to the Char overloaded version of WriteBuffer, with the result being only a single character (`S1[1]`) being written to the stream.

Other Pointer Types

Other typed pointers have been eliminated from the VCL. Typically, they are replaced by the type to which the original pointed. If the pointer type was the parameter to a procedure call, it is typically converted to a var parameter so that the resulting code still passes a reference rather than a copy of the argument. Sometimes, it is useful to change a value type into a class type so that rather than passing a typed pointer, your code passes an object reference.

Migrating Char and String Types

In Developer Studio 2006, the string type maps to the .NET String type, and you can freely access the members of String using a Delphi string type, as demonstrated in the following example:

```
var
  S: string;
begin

  S := 'This is a string';

  // Note the typecast is not necessary.
  // S := System.String(S).PadRight(25);


  // Direct access to string class members
  S := S.PadRight(25);
  S := ('This is a new string').PadRight(25);
```

ANSI Strings and Wide Strings

The biggest difference for strings in Developer Studio 2006 is that the string type is now a Unicode wide string rather than an AnsiString. This simplifies code for some locales, because you no longer need to worry about multibyte character sets. However, you must examine your code for any assumptions about the size of a Char, because it is now two bytes rather than one. You can still use strings with one-byte characters, but you must now declare them

as AnsiString rather than string. The compiler converts between wide and narrow strings if you use an explicit typecast or if you implicitly cast them by assigning to a variable or parameter of the other type.

If your code calls any of the `AnsiXXX` routines for manipulating strings, you may want to change these to the corresponding wide string version of the routine. The `AnsiXXX` routines have (deprecated) overloads that map to the wide versions, and the overloaded routines accept wide strings for their parameters; this avoids implicit conversion back and forth between wide and single-byte strings.

Note: Information can be lost when converting from wide to single-byte characters, therefore, you should avoid downcasting as much as possible.

String Operations

Following the CLR value-type semantics, typically operations on strings return a copy of the string rather than alter the existing string. This may make some code less efficient, because there is more copying going on. For example, consider the following:

```
var
  S: string;
begin
  S := 'This is a string';
  S[3] := 'a';
  S[4] := 't';
```

When compiled using on the Win32 platform, the character substitutions only require a single byte of memory to change each time. In Developer Studio 2006, each substitution results in a copy of the entire string. Because of this, it is a good idea to use a StringBuilder instance when you are manipulating string values. StringBuilder allocates a chunk of unmanaged memory and manipulates the string the way you expect. When you are finished, you can convert the result to a string by calling the ToString method.

Note: The conversion to string from a StringBuilder is a low-cost operation. The string data is not copied again.

Uninitialized Strings

In Developer Studio 2006, an uninitialized string has the value of nil. The compiler will automatically compensate if you compare an uninitialized string with an empty string. That is, if you have a line such as

```
if S <> '' then ...
```

The compiler handles the comparison and treats the uninitialized string as an empty string. However, unlike code compiled on the Win32 platform, other string operations do not automatically treat an uninitialized string like an empty string. This can lead to Null Object exceptions at runtime.

Typecasts

Unlike Delphi for Win32, in Developer Studio 2006, there is no distinction between an explicit typecast and the as operator. In both cases, the cast only succeeds if the variable being cast is really an instance of the type to which you cast it. This means that code which used to work (by casting between incompatible data types) may now generate a runtime exception.

Message Crackers

Perhaps the most common situation where the change to typecasts causes a problem is in the use of the message cracker types. In the VCL on Win32, the `Messages` unit defined a number of record types to represent the parameters

of a Windows message. These records were all the same size, with the fields laid out to extract the information from the Windows message. Thus, you could have the message parameters in one form (say, TMessage), and typecast it to another (say TWMMouse), and extract the information you wanted. This worked because the two types were the same size, and an explicit typecast did not raise an exception when you reinterpreted the type with the cast. Such a reinterpret cast is not allowed in .NET, and the same code would lead to an invalid cast exception in Developer Studio 2006.

To work around this situation, the message cracker types in Developer Studio 2006 are not records at all, but classes. Instead of casting a TMessage value to another type such as TWMMouse, you must instantiate the other type, passing the original TMessage as a parameter. That is, instead of

```
procedure MyFunction(Msg: TMessage);
var
  MouseMsg: TWMMouse;
begin
  if Msg.Msg = WM_MOUSE then
    with Msg as TWMMouse do
    ...
end;
```

you would do something like the following:

```
procedure MyFunction(Msg: TMessage);
var
  MouseMsg: TWMMouse;
begin
  if Msg.Msg = WM_MOUSE then
    with TWMMouse.Create(Msg) do
    ...
end;
```

To convert in the other direction (from a specialized message type to TMessage), you can use the new `UnwrapMessage` function that is declared in the `Messages` unit.

Accessing Protected Members from Classes in Other Units

Another technique that involves what is now an invalid typecast is when you need to access the protected members of a class that is declared in another unit. In Delphi for Win32, you can declare a descendant of the class whose members you want to see:

```
type
  TPeekAtWinControl = class(TWinControl);
```

Then, by casting an arbitrary TWinControl descendant to `TPeekAtWinControl`, you could access the protected methods of TWinControl, because `TPeekAtWinControl` was defined in the same unit.

In general, this technique does not work in Developer Studio 2006, because the arbitrary TWinControl descendant is not, in fact, an instance of `TPeekAtWinControl`. The cast leads to an invalid cast exception at runtime.

Because this is a widely used technique in Win32, the compiler will recognize this pattern and allow it. However, the compiler can't know what assembly a unit will be linked into when it compiles the source code. If the units are linked into assemblies, this technique will fail at runtime with a type exception.

When you need to cross assembly boundaries, one workaround is to introduce an interface that provides access to the protected members in question. Some of the classes in the VCL (TControl, TWinControl, TCustomForm) now

use this technique, and you can find the addition of interfaces to access protected members (IControl, IWinControl, IMDIForm).

Creating and Destroying Objects

Specific language issues with programming in Delphi on the memory-managed .NET platform are explained in the topic Memory Management Issues on the .NET Platform.

Because of differences in the way objects are instantiated and freed, it is not possible to have a `BeforeDestruction` or `AfterConstruction` method on a Developer Studio 2006 class. Any classes that override these methods must be rewritten.

The fact that these methods and the `OldCreateOrder` property do not exist in the VCL on the .NET platform impacts forms and data modules that relied on `OldCreateOrder` being False. The `OnCreate` and `OnDestroy` events now act as if the `OldcreateOrder` property is set to True, and will only be called from the constructor or destructor.

Note: Because `OnDestroy` is called from a destructor, it is not guaranteed to be called – if the application does not call `Free`, the object's destructor is not called, even though it is garbage collected.

Working with the Unmanaged Win32 API

Most of the VCL is designed for working with the Windows API. This is handled in a way analogous to the way `Systems.Windows.Forms` works: The VCL is a managed API that calls into the Windows API, marshaling between the managed structures on the VCL side and the unmanaged types that the Windows API uses. Some units, particularly in the RTL, have been ported so that they sit on top of CLR rather than the Windows API. Such units are more flexible, because they can work with any .NET environment, even those that do not support the Windows operating system (for example, the Compact Framework, Mono, and so on). Units that require the Windows operating system are tagged with the platform directive. In units that are not tagged with the platform directive, any methods or classes that require Windows are tagged with the platform directive.

Isolating Windows Dependencies

In order to maintain relative platform independence in RTL units, some methods functions that rely on Windows have been moved into the `WinUtils` unit. In addition, some classes have been changed to rely more on CLR than Windows.

TObject, Exception, TPersistent, and TComponent, all map directly to classes implemented in the .NET Framework. In this way they integrate more smoothly with other .NET applications. Because the corresponding CLR classes (System.Object, System.Exception, System.Marshal, and System.Component) do not include all the methods that the VCL requires, the missing methods are supplied by Delphi class helper declarations. In most cases, this mechanism is transparent. However, there are a few cases where it requires you to make minor tweaks to your code. For example, with TComponent, FComponentState is now a property of TComponentHelper rather than a true field of TComponent. This means that you can't use the Include and Exclude methods on FComponentState, because when passed a property, they operate on a copy of the property value, which does not alter FComponentState. Thus code such as

```
Exclude(FComponentState, csUpdating);
```

Must be rewritten as

```
FComponentState := FComponentState - [csUpdating];
```

TThread has also been changed to map to the CLR thread object. This means that the Thread handle is no longer an ordinal type, but is rather a reference to the underlying CLR thread object. It also means that TThread no longer

supports a ThreadID, which is not supported by the CLR thread object. If your thread class requires a ThreadID, you should change it to derive from TWin32Thread instead.

Calling the Windows API

Many Windows APIs have changed to use a more managed interface. Often, the types of parameters have changed, typically to eliminate pointers. One common change is the PChar types have been replaced by string or StringBuilder.

When your application calls a Windows API, it is making a call into an unmanaged DLL. Because of this, all parameter values must be marshaled into unmanaged memory, where Windows can work with it, and results are then unmarshalled back into managed memory. In most cases, this marshaling is handled automatically, based on the attributes that have been added to API declarations or type declarations. There are some cases, however, when your code must explicitly handle the marshaling – especially when dealing with a pointer on a structure. To do this marshaling, use the System. Marshal class. Another class that can be very useful when marshaling data to or from unmanaged memory is the BitConverter class. For example, the Marshal class does not include a method to read or write a double value, but it can read or write `Int64` values, which are the same size, and the BitConverter class can convert these to or from doubles:

```
// copy double into unmanaged memory:
Mem := Marshal.AllocHGlobal(SizeOf(Int64));
Marshal.WriteInt64(Mem, BitConverter.DoubleToInt64Bits(DoubleVariable));
...
// copy double from unmanaged memory
DoubleVariable := BitConverter.Int64BitsToDouble(Marshal.ReadInt64(Mem));
```

When using the marshal class, remember that you must always free any unmanaged memory you allocate – the garbage collector does not collect unmanaged memory.

Working with Windows Messages

One of the changes in the way Developer Studio 2006 applications work with Windows is the way message handlers work. The basics of declaring and using messages handlers is the same, but the message-cracker types have changed from records to classes, and you can no longer simply typecast from one message-cracker type to another. Most of this has already been covered in the section on typecasts, but there are a few additional issues that bear mentioning:

- When porting code that sends a message, it is no longer sufficient to declare the message cracker on the stack, fill out its fields, and pass it to a call to `SendMessage`. You must now add a call to create the message cracker, because it is now a class.
- Inside a message handler, you can still call an inherited message handler using the inherited keyword. However, if you do this, you must now be sure that the message cracker type is the same as that in the inherited message handler. For example, if the inherited message handler has a parameter of type TWMMouse, and your message handler only needs TMessage, declaring your message handler to use TMessage and calling inherited will lead to an invalid cast exception at runtime. Thus, if you call the inherited message handler, you must now ensure that your message parameter matches that of the inherited handler.
- If a message has parameters that are pointers to records (or pointers to anything, for that matter), then the corresponding message cracker will have properties that represent those records. It is important to realize, however, that these are properties and not fields. Thus, you can read the fields of the record directly from the property, but if your handler needs to change any field values, you can no longer make assignments directly to the fields of the record. Instead, you must copy the record to a local variable, make your changes, and then assign the result back to the property.

Using Windows messages is somewhat more expensive in Developer Studio 2006, because in addition to the overhead of working with the message queue, there is now the overhead of marshaling values to and from unmanaged memory. This is particularly expensive when a parameter represents a pointer (an object reference or

a pointer to a structure). Such parameters are ultimately converted to a `WPARAM` or `LPARAM` using an IntPtr, which acts as a handle to a block of unmanaged memory that contains a copy of the structure. Object references are converted using a GCHandle. In most cases, the predefined message cracker types handle the marshaling of these parameters, to and from the IntPtr, but if you defining your own messages, you may need to perform your own marshaling. The message cracker classes defined in the `Controls` unit illustrate how to handle these marshaling issues.

The VCL defines and uses a number of private message types. These are, for the most part, defined in the `Controls` unit, and have identifiers of the form `CM_XXX` or `CN_XXX`. Because of the extra overhead in marshaling messages, several of the `CM_XXX` message types have been changed or eliminated, replaced by other mechanisms that are less expensive in the .NET environment. The following table lists the message types that have changed, and how the same task is accomplished in Developer Studio 2006:

Message type	Change
CM_FOCUSCHANGED	Replaced by a protected method (FocusChanged) on TWinControl. Replace message handlers by an override to the FocusChanged method. Instead of sending messages, call FocusChanged using the IWinControl interface.
CM_MOUSEENTER	Meaning of LPARAM has changed. It used to pass an object reference to the child control where the mouse entered – now it passes the index of that child in the FWinControls or FControls list.
CM_MOUSELEAVE	Meaning of LPARAM has changed. It used to pass an object reference to the child control where the mouse exited – now it passes the index of that child in the FWinControls or FControls list.
CM_BUTTONPRESSED	Replaced by a protected method (ButtonPressed) on TSpeedButton. This was only used by TSpeedButton. The CMButtonPressed message handler was replaced by ButtonPressed, which is called directly.
CM_WINDOWHOOK	Retired. TApplication.HookMainWindow and TApplication.UnhookMainWindow are both public methods that can be called directly.
CM_CONTROLLISTCHANGE	Replaced by a protected method (ControlListChange) on TWinControl. Replace message handlers by an override to the ControlListChange method.
CM_GETDATALINK	Replaced by a protected method (GetDataLink) on various data-aware controls. Call this using the new IDataControl interface. When creating your own data-aware control (that does not descend from an existing class in DBCtrls), you must implement IDataControl if the control is to work in a DBCGrid.
CM_CONTROLCHANGE	Replaced by a protected method (GetDataLink) on various data-aware controls. Call this using the new IDataControl interface. When creating your own data-aware control (that does not descend from an existing class in DBCtrls), you must implement IDataControl if the control is to work in a DBCGrid.
CM_CHANGED	Meaning of LPARAM has changed. It used to pass an object reference, now it passes a hash code for the object that changed.
CM_DOCKCLIENT	Replaced by a protected method (DockClient) on TWinControl. Replace message handlers by an override to the DockClient method.
CM_UNDOCKCLIENT	Replaced by a protected method (UndockClient) on TWinControl. Replace message handlers by an override to the UndockClient method.
CM_FLOAT	Replaced by a protected method (FloatControl) on TControl. Replace message handlers by an override to the FloatControl method.
CM_ACTIONUPDATE	Retired. TApplication.DispatchAction was promoted to public, and is called directly rather than using a message.
CM_ACTIONEXECUTE	Retired. TApplication.DispatchAction was promoted to public and is called directly rather than using a message.

Changes to the Threading Model

Sometimes, Windows API calls require the use of the Single Threaded Apartment (STA) model to function properly on some operating systems. For example, on some versions of Windows 98, the `Open` and `Save` dialogs do not work unless your Developer Studio 2006 application uses the Single Threaded Apartment model. Any portion of the VCL that uses COM requires this model.

The threading model is established when the process first starts up. If you are creating an executable, this is easy: just add the `[STAThreadAttribute]` attribute to the line immediately preceding the begin statement in the dpr file. When creating a DLL, you can't force the threading model. However, you can call the CheckThreadingModel procedure in the `SysUtils` unit to raise an exception when the application calls a method that requires a particular threading model.

This restriction is fairly common in .NET. By default, Microsoft Visual Studio adds the STAThreadAttribute attribute to applications it creates.

Migrating Variants

The Variant type is very different in Developer Studio 2006. Whereas the Win32 compiler maps Variant onto the record type that COM uses for Variants, in Developer Studio 2006, a Variant is more general. Any object (which in Developer Studio 2006 is any type) can act be manipulated as a Variant. Thus, in Developer Studio 2006, you could assign a control to a Variant.

The Delphi Variant type is a Delphi language notion that is not CLS compliant. If you are writing code in Developer Studio 2006 that uses Variants, to the outside world will see, these will map to only as System. Object. Thus, to code written in other languages, the flexibility in type conversions that Delphi Variants support provide is not available.

Changes to TVarRec

If your code uses Variants, chances are it should still work. However, because Variants are no longer based on the TVarRec type, any code that works with the internals of a Win32 Variant by getting into the underlying TVarRec record must be rewritten for .NET.

Note: Nearly all of the functions provided by the `Variants` unit are implemented in Developer Studio 2006. If you need to get the VarType of a Variant, you can accomplish this and still maintain platform portable code.

Changes to OLE Variants

The COM Interop layer automatically marshals Objects (and hence Variants). Thus, you can use Developer Studio 2006 Variants with COM. However, when using Developer Studio 2006 Variants with COM, you should restrict the types you assign to the Variant to COM-compatible types.

In Delphi for Win32, the compiler enforces COM restrictions on the kinds of data that can be assigned to an OleVariant. In Developer Studio 2006, OleVariant is simply a synonym for Variant. It does nothing to ensure that the Variant value is a COM-compatible type.

Changes to Custom Variants

Custom Variants are completely different in Developer Studio 2006. Because Variants are just objects, you do not need to do anything at all to create a custom Variant – any class you define is already a Variant type. However, to work well as a custom Variant, it helps to implement some CLR interfaces: IComparable, IConvertible, and ICloneable. The Delphi compiler can use these to implement Variant operations. Even with these interfaces, however, other, arbitrary Variant types, can't be converted into your Variant (class) unless you implement a FromObject method:

```
class function FromObject(AObject: System.Object): TObject; static;
```

FromObject takes an arbitrary source object (the Variant to convert to your class type) and returns the corresponding instance of your class as a TObject.

Working With Resources

Developer Studio 2006 can link Windows resources (res files) into your assemblies. This means that when first porting an application, you do not need to change the way you declare and use resources, and it will still work. In some cases, this is what you want to do anyway. For example, if you use custom cursors, it is simpler to use the Windows API `LoadCursor` function to add the cursor to TScreen.Cursors than to bring in the overhead of using Cursor and then obtaining a handle to the underlying cursor. However, for resources that are not Windows-specific (such as bitmaps, icons, and strings) you will probably want to update to a .NET resources file.

Resource Strings

When you use the resourcestring keyword, Developer Studio 2006 automatically creates the string resources as .NET resources rather than Windows resources. This happens automatically and there is nothing special you need to do. The one thing to watch out for is that you no longer can use the `PResStringRec` type.

Bitmaps

You can convert bitmaps into .NET resources using the ResourceWriter class. The resulting resources file can be linked into your Developer Studio 2006 application, or deployed as a satellite assembly. To use these converted bitmaps, LoadFromResourceName has new overloads for working with .NET resources (and the old version of LoadFromResourceName as well as the LoadFromResourceID method have been deprecated.) Thus, for example, if your bitmaps are in a resources file with a name such as MyResources.en-US.resources, you can load your bitmap as follows

```
MyBitmap.LoadFromResourceName('MyFirstBitmap', 'MyResources', System.
Assembly.GetCallingAssembly);
```

Note that this example assumes the resources are compiled into the assembly that is making the method call that contains this line. If the resources are compiled into a different assembly, you can use System. Assembly.GetAssembly (using a type that is defined in the relevant assembly) or System. Assembly.GetExecutingAssembly (to obtain the currently executing assembly).

Change to TTreeView.OnCompare

The signature for the OnCompare event in the TTreeView class has changed in the VCL for .NET. Existing code will cause a runtime exception when the event handler is called.

In Delphi 7, the signature was:

```
TTVCompareEvent = procedure(Sender: TObject; Node1, Node2: TTreeNode; Data: Integer; var
Compare: Integer) of object;
```

In Delphi for .NET, the new signature is:

```
TTCompareEvent = procedure(Sender: TObject; Node1, Node2: TTreeNode; Data: TTag; var
Compare: Integer) of object;
```

Building Database Applications with ADO.NET

ADO.NET presents a coherent programming model for exposing data access within the .NET Framework. In addition to supporting MS SQL, Oracle, and OLE DB connection components within the .NET Framework, Developer Studio 2006 includes Borland Data Providers for .NET (BDP.NET). BDP.NET supports access to MS SQL, Oracle, DB2, and Interbase. BDP.NET component designers ease the generation and configuration of BDP.NET components.

If you are developing new VCL Forms applications for the .NET Framework, or you are migrating existing Win32 VCL Forms applications to the .NET Framework, Developer Studio 2006 provides continued support for existing Delphi database technologies, such as dbExpress and dbGo.

This section includes conceptual information about how to use Developer Studio 2006 with the ADO.NET architecture, as well as the VCL for .NET database technologies. and how to build a simple ADO.NET project.

In This Section

ADO.NET Overview
Introduces the ADO.NET architecture for building database applications using Developer Studio 2006.

Borland Data Providers for Microsoft .NET
Describes the providers included with Developer Studio 2006.

BDP.NET Data Types
Describes the data types included with Borland Data Provider for .NET.

BDP.NET Component Designers
Introduces component designers and property editors for databases components.

Stored Procedure Overview
Describes how stored procedures can be used within Developer Studio 2006.

VCL for .NET Database Technologies
Describes Delphi database technologies that you can use in VCL.NET applications.

dbExpress Components overview
Introduces dbExpress, a set of thin database connectivity components.

dbGo Components Overview
Introduces dbGo, a library of ADO components to help you develop ADO database applications rapidly.

Getting Started with InterBase Express
Getting Started with InterBase Express

Deploying Database Applications for the .NET Framework
Provides general information about deploying database applications on the .NET Framework.

Building a Windows Forms Database Application
Describes the essential tasks to create an ADO.NET application using Windows Forms and BDP.NET.

Building an ASP.NET Database Application
Describes the essential tasks to create an ADO.NET application using Web Forms and BDP.NET.

ADO.NET Overview

ADO.NET is the .NET programming environment for building database applications based on native database formats or XML data. ADO.NET is designed as a back-end data store for all .NET programming models, including Web Forms, Web Services, and Windows Forms. Use ADO.NET to manage data in the .NET Framework.

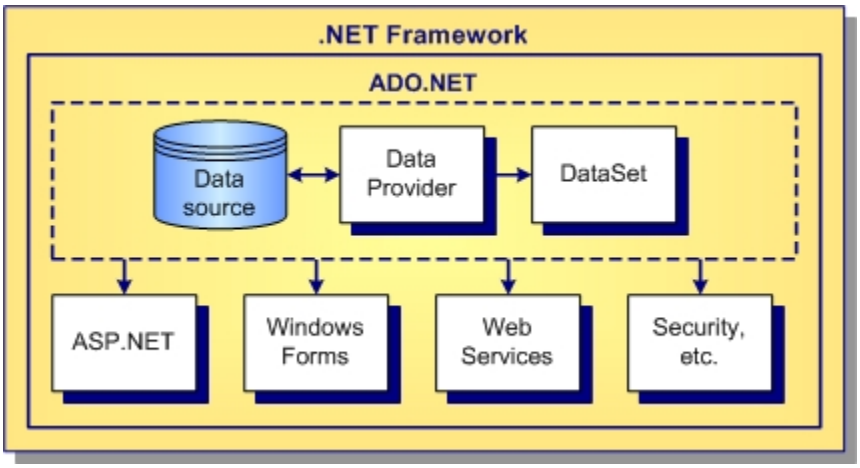
Borland provides tools to simplify rapid ADO.NET development using Borland Data Providers for .NET (BDP.NET). If you are familiar with rapid application development (RAD) and object oriented programming (OOP) using properties, methods, and events, you will find the ADO.NET model for building applications familiar. If you are a traditional database programmer, ADO.NET provides familiar concepts, such as tables, rows, and columns with relational navigation. XML developers will appreciate navigating the same data with nodes, parents, siblings, and children.

This topic discusses the major components of the ADO.NET architecture, how ADO.NET integrates with other programming models in the .NET Framework, and key Developer Studio 2006 functionality to support ADO.NET.

This topic introduces:

- ADO.NET Architecture
- ADO.NET User Interfaces
- BDP.NET Namespace

ADO.NET Architecture



The two major components of the ADO.NET architecture are the Data Provider and the DataSet. The data source represents the physical database or XML file, the Data Provider makes connections and passes commands, and the DataSet represents one or more data sources in memory. For more information about the general ADO.NET model, see the Microsoft .NET Framework SDK documentation.

Data Source

The data source is the physical database, either local or remote, or an XML file. In traditional database programming, the developer typically works with the data source directly, often requiring complex, proprietary interfaces. With ADO.NET, the database developer works with a set of components to access the data source, to expose data, and to pass commands.

Data Providers

Data Provider components connect to the physical databases or XML files, hiding implementation details. Providers can connect to one or more data sources, pass commands, and expose data to the DataSet.

The .NET Framework includes providers for MS SQL, OLE DB, and Oracle. In addition to supporting the .NET providers, this product includes BDP.NET. BDP.NET connects to a number of industry standard databases, providing a consistent programming environment. For more information, see the Borland Data Providers for Microsoft .NET topic.

The TADONETConnector component provides access to .NET DataSets either directly or through BDP.NET. TADONETConnector is the base class for Developer Studio 2006 datasets that access their data using ADO.NET. TADONETConnector descendants include TCustomADONETConnector. TADONETConnector is a descendent of TDataSet.

DataSet

The DataSet object represents in-memory tables and relations from one or more data sources. The DataSet provides a temporary work area or virtual scratch pad for manipulating data. ADO.NET applications manipulate tables in memory, not within the physical database. The DataSet provides additional flexibility over direct connections to physical databases. Much like a typical cursor object supported by many database systems, the DataSet can contain multiple DataTables, which are representations of tables or views from any number of data sources. The DataSet works in an asynchronous, non-connected mode, passing update commands through the Provider to the data source at a later time.

Developer Studio 2006 provides two kinds of DataSets for your use: standard DataSets and typed DataSets. A standard DataSet is the default DataSet that you get when you define a DataSet object implicitly. This type of DataSet is constructed based on the layout of the columns in your data source, as they are returned at runtime based on your Select statement.

Typed DataSets provide more control over the layout of the data you retrieve from a data source. A typed DataSet derives from a DataSet class. The typed DataSet lets you access tables and columns by name rather than collection methods. The typed DataSet feature provides better readability, improved code completion capabilities, and data type enforcement unavailable with standard DataSets. The compiler checks for type mismatches of typed DataSet elements at compile time rather than runtime. When you create a typed dataset, you will see that some new objects are created for you and are accessible through the **Project Manager**. You will notice two files named after your dataset. One file is an XML .xsd file and the other is a code file in the language you are using. All of the data about your dataset, including the table and column data from the database connection, is stored in the .xsd file. The program code file is created based on the XML in the .xsd file. If you want to change the structure of the typed dataset, you can change items in the .xsd file. When you recompile, the program code file is regenerated based on the modified XML.

For more information about DataSets, see the Microsoft .NET Framework SDK documentation.

ADO.NET User Interfaces

ADO.NET provides data access for the various programming models in .NET.

Web Forms

Web Forms in ASP.NET provide a convenient interface for accessing databases over the web. ASP.NET uses ADO.NET to handle data access functions.

.NET and BDP.NET connection components ease integration between Web Forms and ADO.NET. DB Web Controls support both ADO.NET and BDP.NET components, accelerating web application development.

Windows Forms

As an alternative to Web Forms, traditional, native-OS clients can function as a front end to ADO.NET databases.

In Developer Studio 2006 you can provide two types of Windows Forms: a TWinForm object, which is a descendant of TForm and acts as the native .NET Windows Form, and a VCL.NET form.

BDP.NET Namespace

BDP.NET classes are found under the `Borland.Data` namespace.

BDP.NET Namespace

Namespace	Description
Borland.Data.Common	Contains objects common to all Borland Data Providers, including Error and Exceptions classes, data type enumerations, provider options, and Interfaces for building your own Command, Connection, and Cursor classes.
Borland.Data.Provider	Contains key BDP.NET classes like BdpCommand, BdpConnection, BdpDataAdapter, and others that provide the means to interact with external data sources, such as Oracle, DB2, Interbase, and MS SQL Server databases.
Borland.Data.Schema	Contains Interfaces for building your own database schema manipulation classes, as well as a number of types and enumerators that define metadata.

Borland Data Providers for Microsoft .NET

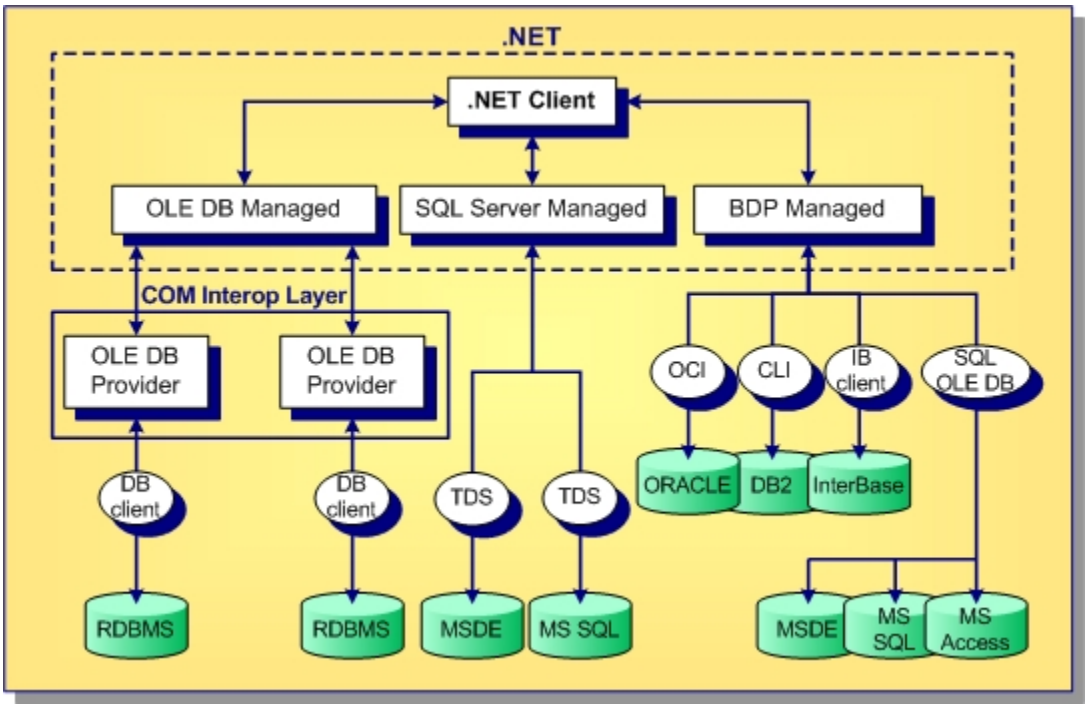
In addition to supporting the providers included in the .NET Framework, Developer Studio 2006 includes Borland Data Providers for Microsoft .NET (BDP.NET). BDP.NET is an implementation of the .NET Provider and connects to a number of popular databases.

This topic includes:

- Data Provider Architecture
- BDP.NET Advantages
- BDP.NET and ADO.NET Components
- Supported BDP.NET Providers
- BDP.NET Data Types
- BDP.NET Interfaces

Data Provider Architecture

Developer Studio 2006 supports the .NET Framework providers and the BDP.NET providers.



BDP.NET provides a high performance architecture for accessing data sources without a COM Interop layer.

The architecture exposes a set of interfaces for third-party integration. You can implement these interfaces for your own database to provide designtime, tools, and runtime data access integration into the Borland IDE. BDP.NET-managed components communicate with these interfaces to accomplish all basic data access functionality. These interfaces were implemented to wrap database-specific native client libraries by way of Platform Invoke (P/Invoke) services. Depending on the availability of managed database clients, you can implement a fully-managed provider underneath BDP.NET.

The database-specific implementation is wrapped into an assembly and the full name of the assembly is passed to the BdpConnection component as part of the connection string. Depending on the Assembly entry in the ConnectionString property, BDP.NET dynamically loads the database-specific provider and consumes the

implementation for ISQLConnection, ISQLCommand, and ISQLCursor. This allows you to switch applications from one database to another just by changing the ConnectionString property to point to a different provider.

BDP.NET Advantages

BDP.NET provides a number of advantages:

- Unified programming model applicable to multiple database platforms
- High performance data-access architecture
- Open architecture, which supports additional databases easily
- Portable code to write once and connect to any supported databases
- Consistent data type mapping across databases where applicable
- Logical data types mapped to .NET native types
- No need for a COM Interop layer, unlike OLE DB
- Lets you view live data as you design your application
- Extends ADO.NET to provide interfaces for metadata services, schema creation, and data migration
- Rich set of component designers and tools to speed database application development

Developer Studio 2006 extends .NET support to additional database platforms, providing a consistent connection architecture and data type mapping.

BDP.NET and ADO.NET Components

The DataSet is an in-memory representation of one or more DataTables. Each DataTable in a DataSet consists of DataColumns and DataRows. The DataSet is generated as a result of an SQL query that you supply to the provider. You can navigate the DataSet like you would any standard relational table. BDP.NET providers encapsulate implementation details for each database type, yet allow you to customize your SQL statements and manage the result sets with complete flexibility.

BDP.NET includes several designtime components that you can place onto a Windows Form or Web Form. A set of designers are also provided to help you build your data connections, DataSets, relations, and other elements.

The primary components that are most useful, particularly if you decide to implement your own database-specific provider, are:

- BdpConnection—establishes a database connection
- BdpCommand—includes a set of methods and properties for SQL and stored procedure execution
- BdpDataReader—retrieves data
- BdpParameter—supports runtime parameter binding
- BdpTransaction—supports transaction control
- BdpDataAdapter—provides and resolves data
- BdpCopyTable—migrates table structures, primary keys, and data
- ISQLMetaData—retrieves metadata
- ISQLSchemaCreate—includes methods for creating, dropping, and altering database objects

For more information, click on the link for each component, or search for the components in the API reference documentation in this Help.

Supported BDP.NET Providers

BDP.NET includes providers for a number of industry-standard databases. These are shown in the following table, along with their corresponding namespaces.

Database	Namespace
InterBase	`Borland.Data.Interbase`
Oracle	`Borland.Data.Oracle`
IBM DB2	`Borland.Data.Db2`
Microsoft SQL Server	`Borland.Data.Mssql`
Microsoft Access	`Borland.Data.Msacc`
Sybase	`Borland.Data.Sybase`

The BDP.NET components, metadata access, and designers are defined under the following namespaces:

- `Borland.Data.Provider`
- `Borland.Data.Common`
- `Borland.Data.Schema`
- `Borland.Data.Design`

BDP.NET Data Types

BDP.NET maps SQL data types to .NET Framework data types, eliminating the need for you to learn a database-specific type system. Every attempt has been made to implement consistent type mappings across database types, allowing you to write one set of source that you can run against multiple databases. You can achieve a similar effect with the .NET Framework data providers by communicating with their interfaces directly and by using untyped ancestors. However, once you use strongly typed accessors, your application becomes less portable. BDP.NET does not support any database-specific typed accessors. For more information, see the BDP.NET Data Types topic.

BDP.NET Interfaces

You can extend BDP.NET to support other DBMSs by implementing a subset of the .NET Provider interface. BDP.NET generalizes much of the functionality required to implement data providers. While the .NET Framework gives you the capabilities to create individual data providers for each data source, Borland has simplified the task by offering a generalized set of capabilities. Instead of building separate providers, along with corresponding DataAdapters, DataReaders, Connection objects, and other required objects, you can implement a set of BDP.NET interfaces to build your own data source plug-ins to the Borland Data Provider.

Building plug-ins is a much easier task than building a completely new data provider. You build an assembly that contains the namespace for your provider, as well as classes that encapsulate provider-specific functionality. Much of the functionality you need to connect to, execute commands against, and retrieve data from your data source has already been defined in the Borland Data Provider interfaces.

BDP.NET Data Types

BDP.NET data types map to .NET logical types. Dependant upon the database, BDP.NET data types map to native data types. Where applicable, BDP.NET provides:

- Consistent data type mapping across databases.
- Logical data types mapped to .NET native types.

BDP.NET and .NET Framework

The DataSet class within ADO.NET uses .NET Framework data types. BDP.NET data types logically map .NET data types for supported databases. During designtime, you can use BDP.NET logical types, which will map to the appropriate native type.

Data Types

The .NET Framework includes a wide range of logical data types. BDP.NET inherits logical data types, providing built-in mappings to supported databases. BDP.NET supports logical data type mappings for DB2, InterBase, MS SQL, MSDE, and Oracle.

DB2

BDP.NET supports the following DB2 type mappings.

DB2 Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
BIGINT	Int64	NA	Int64
INTEGER	Int32	NA	Int32
DOUBLE	Double	NA	Double
FLOAT	Float	NA	Single
REAL	Float	NA	Single
DATE	Date	NA	DateTime
TIME	Time	NA	DateTime
TIMESTAMP	Datetime	NA	DateTime
NUMERIC	Decimal	NA	Decimal
DECIMAL	Decimal	NA	Decimal
BLOB	Blob	stBinary	Byte[]
CLOB	Blob	stMemo	Char[]

InterBase

BDP.NET supports the following InterBase type mappings.

InterBase Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
INTEGER	Int32	NA	Int32
FLOAT	Float	NA	Single
DOUBLE	Double	NA	Double
BLOB Sub_Type 0	Blob	stBinary	Byte[]
BLOB Sub_Type 1	Blob	stMemo	Char[]
TIMESTAMP	Datetime	NA	DateTime

MS SQL and MSDE

BDP.NET supports the following MS SQL and MSDE type mappings.

MSSQL Type	Bdp Type	BdpSubType	System.Type
BIGINT	Int64	NA	Int64
INT	Int32	NA	Int32
SMALLINT	Int16	NA	Int16
TINYINT	Int16	NA	Int16
BIT	Boolean	NA	Boolean
DECIMAL	Decimal	NA	Decimal
NUMERIC	Decimal	NA	Decimal
MONEY	Decimal	NA	Decimal
SMALLMONEY	Decimal	NA	Decimal
FLOAT	Double	NA	Double
REAL	Float	NA	Single
DATETIME	DateTime	NA	DateTime
SMALLDATETIME	DateTime	NA	DateTime
CHAR	String	stFixed	String
VARCHAR	String	NA	String
TEXT	Blob	stMemo	Char[]
BINARY	VarBytes	NA	Byte[]
VARBINARY	VarBytes	NA	Byte[]
IMAGE	Blob	stBinary	Byte[]
TIMESTAMP	VarBytes	NA	Byte[]

UNIQUEIDENTIFIER	Guid	NA	Guid

Oracle

BDP.NET supports the following Oracle type mappings.

Oracle Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
NCHAR	String	stFixed	String
VARCHAR	String	NA	String
NVARCHAR	String	NA	String
VARCHAR2	String	NA	String
NVARCHAR2	String	NA	String
NUMBER	Decimal	NA	Decimal
DATE	Date	NA	DateTime
BLOB	Blob	stHBinary	Byte[]
CLOB	Blob	stHMemo	Char[]
LONG	Blob	stMemo	Char[]
LONG RAW	Blob	stBinary	Byte[]
BFILE	Blob	stBFile	Char[]
ROWID	String	NA	String

Sybase

BDP.NET supports the following Sybase type mappings.

Sybase Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
INT	Int32	NA	Int32
SMALLINT	Int16	NA	Int16
TINYINT	Int16	NA	Int16
DOUBLE PRECISION	Float	NA	Single
FLOAT	Float	NA	Single
REAL	Float	NA	Single
NUMERIC	Decimal	NA	Decimal
DECIMAL	Decimal	NA	Decimal
SMALLMONEY	Decimal	NA	Decimal
MONEY	Decimal	NA	Decimal

SMALLDATETIME	DateTime	NA	DateTime
DATETIME	DateTime	NA	DateTime
IMAGE	Blob	stBinary	Byte[]
TEXT	Blob	stMemo	Char[]
BIT	Boolean	NA	Boolean
TIMESTAMP	VarBytes	NA	Byte[]
BINARY	Bytes	NA	Byte[]
VARBINARY	VarBytes	NA	Byte[]
SYSNAME	String	NA	String

BDP.NET Component Designers

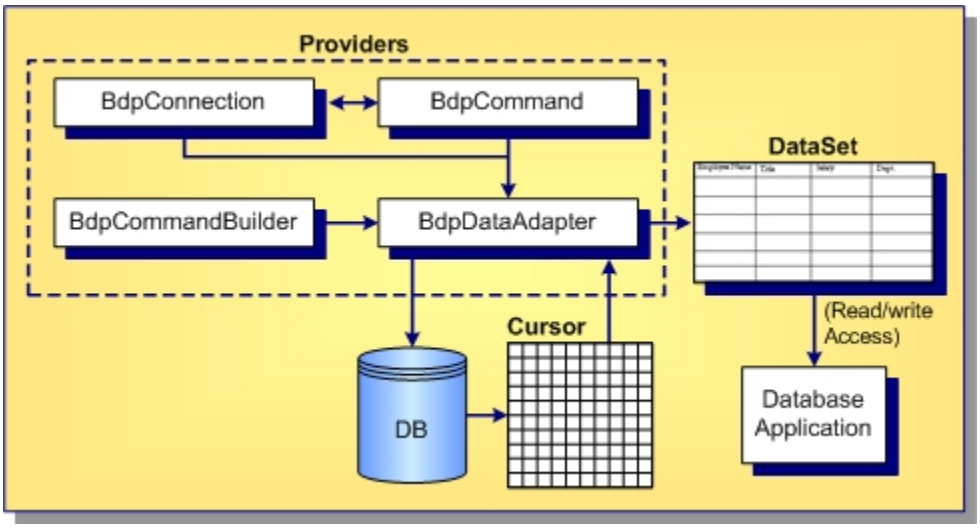
Almost all distributed applications revolve around reading and updating information in databases. Different applications you develop using ADO.NET have different requirements for working with data. For instance, you might develop a simple application that displays data on a form. Or, you might develop an application that provides a way to share data information with another company. In any case, you need to have an understanding of certain fundamental concepts about the data approach in ADO.NET.

Using these designers, you can work efficiently to access, expose, and edit data through database server-specific schema objects like tables, views, and indexes. These designers allow you to use these schema objects to connect to a variety of popular databases, and perform database operations in a consistent and reliable way.

This topic includes:

- Component Designer Relationships
- Connection Editor
- Command Text Editor
- Stored Procedure Dialog Box
- Generate DataSets
- Configure Data Adapter
- Data Explorer

Component Designer Relationships



The major elements of the database component designers include:

- The **Connection Editor** to define a live connection to a data source
- The **Command Text Editor**to construct command text for command components
- The **Configure Data Adapter** to set up commands for a data adapter
- The **Stored Procedure Dialog box** to view and specify values for Input or InputOutput parameters for use with command components
- The **Generate Dataset** to build custom datasets

- The **Data Explorer** to browse database server-specific schema objects and use drag-and-drop techniques to automatically populate data from a data source to your Delphi for .NET project

Connections Editor

The **Connections Editor** manages connection strings and database-specific connection options. Using the **Connections Editor** you can add, remove, delete, rename, and test database connections. Changes to the connection information are saved into the BdpConnections.xml file, where they are accessed whenever you need to create a new connection object. Once you have chosen a particular connection, the **Connections Editor** generates the connection string and any connection options, then assigns them to the ConnectionString and ConnectionOptions properties, respectively.

Display the **Connections Editor** dialog box by dragging the BdpConnection component from the **Tool Palette** onto the form, and then clicking the component designer verb at the bottom of the **Object Inspector**.

Command Text Editor

The **Command Text Editor** can be used to construct the command text for command components that have a CommandText property. A multi-line editing control in the editor lets you manually edit the command or build the command text by selecting tables and columns. Display the **Command Text Editor** dialog box by dragging a BdpCommand component from the **Tool Palette** onto the form, and clicking the designer verb at the bottom of the **Object Inspector**.

The **Command Text Editor** is a simplified version of a SQL builder capable of generating SQL for a single table. The database objects are filtered by the SchemaName property set in the BdpCommand and only tables that are part of that schema are used. If there is no SchemaName listed, all of the available objects for the current login user are listed. The QuoteObjects setting for the ConnectionOptions property determines whether the objects are quoted with the database-specific quote character or not. This is important, for instance, when retrieving tables from databases that allow table names to include spaces.

To populate the Tables and Columns list boxes with items and build SQL statements, you must have defined a live BdpConnection. Otherwise, data cannot be retrieved. The **Command Text Editor** allows you to choose table and column names from a list of available tables and columns. Using this information, the editor generates a SQL statement. To generate the SQL, the editor uses an instance of the BdpCommandBuilder. When you request optimized SQL, the editor uses index information to generate the WHERE clause for SELECT, UPDATE, and DELETE statements; otherwise, non-BLOB columns and searchable columns form the WHERE clause.

When the SQL is generated, the BdpCommand. CommandText property is set to the generated SQL statement.

Stored Procedure Dialog Box

The **Stored Procedure** dialog box is used to view and enter Input and InputOutput parameters for a stored procedure and to execute the stored procedure. Display the **Stored Procedure** dialog box by dragging a BdpCommand component from the **Tool Palette** onto the form, setting the CommandType property for the BdpCommand component to **StoredProcedure**, and clicking the **Command Text Editor** designer verb at the bottom of the **Object Inspector**.

The **Stored Procedure** dialog box lets you select a stored procedure from a list of available stored procedures, which is determined by the BdpConnection specified in the Connection property for the BdpCommand component. When you select a stored procedure, the dialog box displays the parameters associated with the stored procedure, and the parameter metadata for the selected parameter. You can specify values for Input or InputOutput parameters and execute the stored procedure. If the stored procedure returns results, such as Output parameters, InputOutput parameters, return values, cursor(s) returned, they are all populated into a DataGrid in the bottom of the dialog box when the stored procedure is executed. After the CommandText, Parameters, and ParameterCount properties are

all set for the BdpCommand, the stored procedure can be executed at runtime by making a single call to ExecuteReader or ExecuteNonQuery.

Generate DataSets

The **Generate Dataset** designer is used to build a DataSet. Using this tool results in strong typing, cleaner code, and the ability to use code completion. A DataSet is first derived from the base DataSet class and then uses information in an XML Schema file (an .xsd file) to generate a new class. Information from the schema (tables, columns, and so on) is generated and compiled into this new dataset class as a set of first-class objects and properties. Display this dialog box by dragging a BdpDataAdapter component from the **Tool Palette** onto the form, and clicking the component designer verb at the bottom of the **Object Inspector**. If this component is not displayed, choose **Component** ▸ **Installed .NET Components** to add it to the **Tool Palette**.

Configure Data Adapter

The **Configure Data Adapter** designer is used to generate SELECT, INSERT, UPDATE, and DELETE SQL statements. After successful SQL generation, the **Configure Data Adapter** designer creates new BdpCommand objects and adds them to the BdpDataAdapterSelectCommand, DeleteCommand, InsertCommand, and UpdateCommand properties.

After successful SQL SELECT generation, you can preview data and generate a new DataSet. You can also use an existing DataSet to populate a new DataTable. If you create a new DataSet, it will be added automatically to the designer host. You can also generate Typed DataSets.

Data Adapters are an integral part of the ADO.NET managed providers. Essentially, Adapters are used to exchange data between a data source and a dataset. This means reading data from a database into a DataSet, and then writing changed data from the DataSet back to the database. A Data Adapter can move data between any source and a DataSet. Display the **Configure Data Adapter** dialog box by dragging a BdpDataAdapter component from the **Tool Palette** onto the form, and clicking the component designer verb at the bottom of the **Object Inspector**.

Data Explorer

The **Data Explorer** is a hierarchical database browser and editing tool. The **Data Explorer** is integrated into the IDE and can also be run as a standalone executable. To access the **Data Explorer** within the IDE, choose **View** ▸ **Data Explorer**. Use the context menus in the **Data Explorer** to perform the following tasks:

- Manage database connections—add a new connection, modify, delete, or rename your existing connections
- Browse database structure and data—expand and open provider nodes to browse database server-specific schema objects including tables, views, stored procedure definitions, and indexes
- Add and modify tables—specify the data structure for a new table, or add or remove columns, and alter column information for an existing table
- View and test stored procedure parameters—specify values for Input or InputOutput parameters and execute the selected stored procedure
- Migrate data—migrate table schema and data of one or more tables from one provider to another
- Drag-and-drop schema objects onto forms to simplify application development—drag tables or stored procedures onto your application form for the .NET Framework to add connection components and automatically generate connection strings

The **Data Explorer** provides connectivity to several industry-standard databases, and can be extended to connect to other popular databases. The **Data Explorer** uses the ISQLDataSource interface to get a list of available providers, database connections, and schema objects that are supported by different providers. The list of available providers is persisted in the `BdpDataSources.xml` file, and the available connections are persisted in the `BdpConnections.xml` file. Once you have chosen a provider the ISQLMetadata interface is used to retrieve

metadata and display a read-only tree view of database objects. The current implementation provides a list of tables, views, and stored procedures for all BDP.NET-supported databases.

The **Data Explorer** lets you create new tables, alter or drop existing tables, migrate data from multiple tables from one provider to another, and copy and paste individual tables across BDP-supported databases. For all these operations, the **Data Explorer** calls into the ISQLSchemaCreate implementation of the provider.

Additionally, the **Data Explorer** can be used to drag data from a data source to any Developer Studio 2006 project for the .NET framework. Dragging a table onto a form adds BdpConnection and BdpDataAdapter components to your application and automatically configures the BdpDataAdapter for the given table. Dragging a stored procedure onto a form adds BdpConnection and BdpCommand components to your application, and sets the CommandType property of the BdpCommand object to StoredProcedure.

Stored Procedure Overview

All relational databases have certain features in common that allow applications to store and manipulate data. A stored procedure is a self-contained program written in a language specific to the database system. A stored procedure typically handles frequently repeated database-related tasks, and is especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Stored procedures are typically stored on the database server.

Calling a stored procedure is similar to invoking a SQL command, and Developer Studio 2006 provides support for using stored procedures in much the same ways as it supports editing and using SQL command text.

Stored procedures can enhance your database applications in the following ways: improve the performance, security, and reliability of your applications.

- Performance—stored procedures can improve the performance of a database application by taking advantage of the server's usually greater processing power and speed, and reducing network traffic by moving processing to the server. Also, the compiled SQL used in a stored procedure executes faster typically than standard SQL command text.
- Security—by creating a layer between clients and the database, stored procedures can enhance security for your data. You don't need to grant database permissions to individual users. Instead, you can grant users permission to execute a stored procedure independently of underlying table permissions.
- Reliability—stored procedures help to centralize code, which makes it easier to isolate and troubleshoot problems. Also, stored procedures allow you to move business logic which is inherent to the database into the database, thus making it available from all clients regardless of the language they are written in.

When you use BDP.NET, the **Command Text Editor** and the **Data Explorer** both provide the ability to view your stored procedure parameters, specify input parameters, and execute your stored procedures as you design your application.

VCL for .NET Database Technologies

In most cases, BDP.NET provides the best database connectivity solution for your .NET applications. However, if you are developing new VCL Forms applications for the .NET Framework, or you are migrating existing Win32 VCL Forms applications to the .NET Framework, Developer Studio 2006 provides continued support for existing Delphi database technologies.

Developer Studio 2006 provides a migration path from Delphi database technologies running strictly on Win32 clients to the .NET Framework. In addition to being able to build new database applications using ADO.NET and BDP.NET, you can migrate existing database applications to take advantage of .NET capabilities. The Delphi database technologies now supported by Developer Studio 2006 include:

- dbExpress.NET
- DataSnap .NET Client (DCOM)
- IBX.NET (InterBase for .NET)
- BDE.NET
- dbGo

Building .NET Applications with dbExpress.NET

Developer Studio 2006 includes a .NET version of dbExpress. This set of components provide comparable functionality as the dbExpress components for Win32, but updated to run on VCL Forms on the .NET Framework. dbExpress for .NET provides the same lightweight client capability and unidirectional dataset that is available in previous versions of the product.

Building .NET Applications with the DataSnap .NET Client (DCOM)

Developer Studio 2006 provides the means to use the DataSnap (DCOM) client to connect to databases in three-tier applications.

Building .NET Applications with IBX.NET

Developer Studio 2006 provides you with access to InterBase databases, by way of InterBase Express controls, in addition to the standard BDP.NET data adapter or the .NET Framework's ADO.NET providers. IBX.NET controls allow you to connect to InterBase databases, access tables, etcetera.

Building .NET Applications with BDE.NET

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases.

You can connect your Developer Studio 2006 database applications to BDE-supported databases, such as Paradox and dBase.

Building .NET Applications with dbGo

Developer Studio 2006 includes a .NET version of dbGo. This set of components provides comparable functionality as the dbGo components for Win32, but updated to run on VCL Forms on the .NET Framework. dbGo for .NET provides the same powerful and logical object model that is available in previous versions of the product.

dbExpress Components overview

dbExpress is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, dbExpress provides a driver that adapts the server-specific software to a set of uniform dbExpress interfaces. When you deploy a database application that uses dbExpress, you include a DLL (the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets.

dbExpress connections, tables, views, and stored procedures that show up in a data tree view support drag & drop with native and managed vcl forms.

Connection Strings

The ConnectionString property in dbExpress allows you to pass all database options and connection information (database, username, password) by means of a single connection string. This feature also allows you to introduce new properties to your drivers in the middle of a release by changing an interface.

You can load the ConnectionProperties in the dbxconnections.ini for the current connectionName by rightClicking on the connection and selecting the appropriate menu item. This will create a Parameters item (Parameters ['ConnectionString']) that contains all of the connection properties in the inifile. This way you can add new properties to the dbxconnections.ini file, and you don't have type the whole string in yourself.

There is also a 'Clear Connection String' menu item off the SqlConnection rightclick menu, which appears whenever the ConnectionString property is set.

dbExpress Connections

An overloaded connect() method, with no arguments is part ofthe SqlConnection class, and this method has been implemented for all of the drivers.

To use this property, you must call SqlConnection setOption, passing in the connection string to be used, prior to calling connect() without any arguments.

Each driver will extract the valid properties for the driver from the connection string. The database name, username, and password must be included in the connection string, if they are required to connect. Once the connection string has been set, you can call the new connect() method without arguments.

If an invalid parameter type (e.g. setting Rolename for Sybase) is included in the connection string, no error is returned when setting the connection string. Otherwise, dbExpress will raise an error if properties needed to connect are invalid or missing.

Connection Strings for VCL Components

To use the ConnectionString property with VCL, you only need to add a setting to SqlConnection.Parameters. A following setting is a sample for Interbase:

```
ConnectionString=Database=c:\Program Files\Borland\InterBase\gds\examples\database
\employee.gdb,
User_Name=sysdba,Password=masterkey,SqlDialect=3,BlobSize=-1,CommitRetain=True,
ServerCharSet=ASCII
```

Note: None of the options set in the connection string need to be contained in the SqlConnection.Parameters. If they are included they will be ignored.

dbExpress Components

The **dbExpress** section of the **Tool Palette** contains the following components that use **dbExpress** to access database information:

Component	Function
TSQLConnection	Encapsulates a dbExpress connection to a database server
TSQLDataSet	Represents any data available through **dbExpress**, or sends commands to a database accessed through **dbExpress**
TSQLQuery	A query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any
TSQLTable	A table-type dataset that represents all of the rows and columns of a single database table
TSQLStoredProc	A stored procedure-type dataset that executes a stored procedure defined on a database server
TSQLMonitor	Intercepts messages that pass between an SQL connection component and a database server and saves them in a string list
TSimpleDataSet	A client dataset that uses an internal **TSQLDataSet** and **TDataSetProvider** for fetching data and applying updates

dbGo Components Overview

dbGo provides the developers with a powerful and logical object model for programmatically accessing, editing, and updating data from a wide variety of data sources through Microsoft ADO system interfaces. The most common usage of dbGo is to query a table or tables in a relational database, retrieve and display the results in an application, and perhaps allow users to make and save changes to the data.

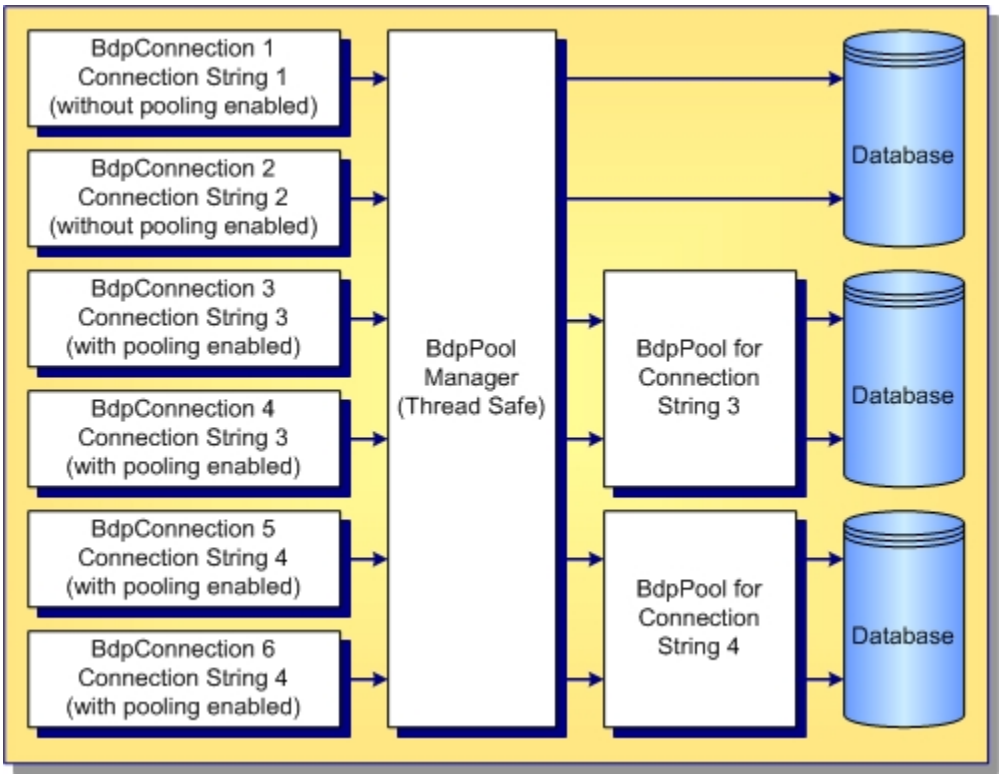
The ADO layer of an ADO-based application consists of the latest version of Microsoft ADO, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional. Microsoft Data Access Components (MDAC) 2.1 or later contains these needed elements. Developer Studio 2006 supports MDAC 2.8.

The **dbGo** section of the **Tool Palette** contains the following components that use **dbGo** to access database information:

Component	Function
TADOConnection	Encapsulates a dbGo connection to a database server
TADODataSet	Represents any data available through **dbGo**, or sends commands to a database accessed through **dbGo**
TADOQuery	A query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any, from an ADO data store
TADOTable	A table-type dataset that represents all of the rows and columns of a single database table
TADOStoredProc	A stored procedure-type dataset that executes a stored procedure defined on a database server
TADOCommand	Represents the ADO Command object, which is used for issuing commands against a data store accessed through an ADO provider
TADODataSet	Represents a dataset retrieved from an ADO data store
TRDSConnection	Exposes the functionality of the RDS DataSpace object

BDP Connection Pooling Overview

You can use the connection pooling options to save connection time by using a connection from an existing pool. When you are using BDP, all connections go through the BDP Pool Manager, even if pooling is not enabled for your connection. For each connection, you can specify: Pooling (enabled or disabled), Minimum Pool Size, Maximum Pool Size, whether connection requests should Grow On Demand, and the number of seconds before a Connection Timeout (or number of seconds for Connection Lifetime).



As shown in the diagram above, the BDP Pool Manager creates a separate pool for each unique connection string. The following connection options are available:.

Options	Function
MinPoolSize	Specifieds the minimum number of connections that will be maintained in the connection pool.
MaxPoolSize	Determines the maximum number of connections in the connection pool. The default maximum size is 100. If GrowOnDemand is False and MaxPoolSize is reached, subsequent connection requests will throw an exception.
GrowOnDemand	Specifies whether the new connection request should grow on demand after a pool reaches the MaxPool Size. Connections that grow on demand will not be returned to the connection pool. Instead, they will be released on BdpConnection.Close().
ConnectionLifetime (Timeout)	Determines the life time of a pooled connection. When a connection returns to the pool, its lifetime is checked to see if it has expired. If it has, then the connection is released instead of returned to the pool. The ConnectionLifetime value is in seconds, and the default is 0.

Getting Started with InterBase Express

InterBase Express (IBX) is a set of data access components that provide a means of accessing data from InterBase databases. The InterBase Administration Components, which require InterBase 6, are described after the InterBase data access components.

IBX components

The following components are located on the InterBase tab of the component palette.

Icon	Component Name	Description
	TIBTable	A dataset component that encapsulates a database table.
	TIBQuery	Executes an InterBase SQL statement.
	TIBStoredProc	Encapsulates a stored procedure on a database server.
	TIBDatabase	Encapsulates an InterBase database connection.
	TIBTransaction	Provides discrete transaction control over a one or more database connections in a database application.
	TIBUpdateSQL	Provides an object for updating read-only datasets when cached updates are enabled.
	TIBDataSet	Executes InterBase SQL statements.
	TIBSQL	Provides an object for executing an InterBase SQL statement with minimal overhead.
	TIBDatabaseInfo	Returns information about the attached database.
	TIBSQLMonitor	Monitors dynamic SQL passed to the InterBase server.
	TIBExtract	Fetches metadata from an InterBase server.
	TIBCustomDataSet	The base class for all datasets that represent data fetched using InterBase Express.

Though they are similar to BDE components in name, the IBX components are somewhat different. For each component with a BDE counterpart, the sections below give a discussion of these differences.

There is no simple migration from BDE to IBX applications. Generally, you must replace BDE components with the comparable IBX components, and then recompile your applications. However, the speed you gain, along with the access you get to the powerful InterBase features make migration well worth your time.

IBDatabase

Use a TIBDatabase component to establish connections to databases, which can involve one or more concurrent transactions. Unlike BDE, IBX has a separate transaction component, which allows you to separate transactions and database connections.

To set up a database connection:

1. Drop an IBDatabase component onto a form or data module.
2. Fill out the DatabaseName property. For a local connection, this is the drive, path, and filename of the database file. Set the Connected property to true.
3. Enter a valid username and password and click OK to establish the database connection.

Warning: Tip: You can store the username and password in the IBDatabase component's Params property by setting the LoginPrompt property to false after logging in. For example, after logging in as the system administrator and setting the LoginPrompt property to false, you may see the following when editing the Params property:

```
user_name=sysdba
password=masterkey
```

IBTransaction

Unlike the Borland Database Engine, IBX controls transactions with a separate component, TIBTransaction. This powerful feature allows you to separate transactions and database connections, so you can take advantage of the InterBase two-phase commit functionality (transactions that span multiple connections) and multiple concurrent transactions using the same connection.

Use an IBTransaction component to handle transaction contexts, which might involve one or more database connections. In most cases, a simple one database/one transaction model will do.

To set up a transaction:

1. Set up an IBDatabase connection as described above.
2. Drop an IBTransaction component onto the form or data module
3. Set the DefaultDatabase property to the name of your IBDatabase component.
4. Set the Active property to true to start the transaction.

IBX dataset components

There are a variety of dataset components from which to choose with IBX, each having their own characteristics and task suitability:

IBTable

Use an TIBTable component to set up a live dataset on a table or view without having to enter any SQL statements.

IBTable components are easy to configure:

1. Add an IBTable component to your form or data module.
2. Specify the associated database and transaction components.
3. Specify the name of the relation from the TableName drop-down list.
4. Set the Active property to true.

IBQuery

Use an TIBQuery component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables.

IBQuery components provide a read-only dataset, and adapt well to the InterBase client/server environment. To set up an IBQuery component:

1. Set up an IBDatabase connection as described above.
2. Set up an IBTransaction connection as described above.

3 Add an IBQuery component to your form or data module.
4 Specify the associated database and transaction components.
5 Enter a valid SQL statement for the IBQuery's SQL property in the String list editor.
6 Set the Active property to true

IBDataSet

Use an TIBDataSet component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables. IBDataSet components are similar to IBQuery components, except that they support live datasets without the need of an IBUpdateSQL component.

The following is an example that provides a live dataset for the COUNTRY table in employee.gdb:

1 Set up an IBDatabase connection as described above.
2 Specify the associated database and transaction components.
3 Add an IBDataSet component to your form or data module.
4 Enter SQL statements for the following properties: SelectSQL, RefreshSQL, ModifySQL, DeleteSQL, InsertSQL. See the following table for example SQL statements.
5 Set the Active property to true.

Sample SQL statements

Property	SQL Statement
SelectSQL	`SELECT Country, Currency FROM Country`
RefreshSQL	`SELECT Country, Currency FROM Country WHERE Country = :Country`
ModifySQL	`UPDATE Country SET Country = :Country, Currency = :Currency WHERE Country = :Old_Country`
DeleteSQL	`DELETE FROM Country WHERE Country = :Old_Country`
InsertSQL	`INSERT INTO Country (Country, Currency) VALUES (:Country, :Currency)`

Note: **Note:** Parameters and fields passed to functions are case-sensitive in dialect 3. For example,

```
FieldByName(EmpNo)
```

would return nothing in dialect 3 if the field was 'EMPNO'.

IBStoredProc

Use TIBStoredProc for InterBase executable procedures: procedures that return, at most, one row of information. For stored procedures that return more than one row of data, or "Select" procedures, use either IBQuery or IBDataSet components.

IBSQL

Use an TIBSQL component for data operations that need to be fast and lightweight. Operations such as data definition and pumping data from one database to another are suitable for IBSQL components.

In the following example, an IBSQL component is used to return the next value from a generator:

1 Set up an IBDatabase connection as described above.
2 Put an IBSQL component on the form or data module and set its Database property to the name of the database.
3 Add an SQL statement to the SQL property string list editor, for example:

```
SELECT GEN_ID(MyGenerator, 1) FROM RDB$DATABASE
```

IBUpdateSQL

Use an TIBUpdateSQL component to update read-only datasets. You can update IBQuery output with an IBUpdateSQL component:

1. Set up an IBQuery component as described above.
2. Add an IBUpdateSQL component to your form or data module.
3. Enter SQL statements for the following properties: DeleteSQL, InsertSQL, ModifySQL, and RefreshSQL.
4. Set the IBQuery component's UpdateObject property to the name of the IBUpdateSQL component.
5. Set the IBQuery component's Active property to true.

IBSQLMonitor

Use an TIBSQLMonitor component to develop diagnostic tools to monitor the communications between your application and the InterBase server. When the TraceFlags properties of an IBDatabase component are turned on, active IBSQLMonitor components can keep track of the connection's activity and send the output to a file or control.

A good example would be to create a separate application that has an IBSQLMonitor component and a Memo control. Run this secondary application, and on the primary application, activate the TraceFlags of the IBDatabase component. Interact with the primary application, and watch the second's memo control fill with data.

IBDatabaseInfo

Use an TIBDatabaseInfo component to retrieve information about a particular database, such as the sweep interval, ODS version, and the user names of those currently attached to this database.

For example, to set up an IBDatabaseInfo component that displays the users currently connected to the database:

1. Set up an IBDatabase connection as described above.
2. Put an IBDatabaseInfo component on the form or data module and set its Database property to the name of the database.
3. Put a Memo component on the form.
4. Put a Timer component on the form and set its interval.
5. Double click on the Timer's OnTimer event field and enter code similar to the following:

```
[Delphi]
Memo1.Text := IBDatabaseInfo.UserNames.Text;   //  Delphi example
```

```
[C++]
Memo1->Text = IBDatabaseInfo->UserNames->Text; //  C++ example
```

IBEvents

Use an *IBEvents* component to register interest in, and asynchronously handle, events posted by an InterBase server.

To set up an IBEvents component:

1. Set up an IBDatabase connection as described above.
2. Put an IBEvents component on the form or data module and set its Database property to the name of the database.

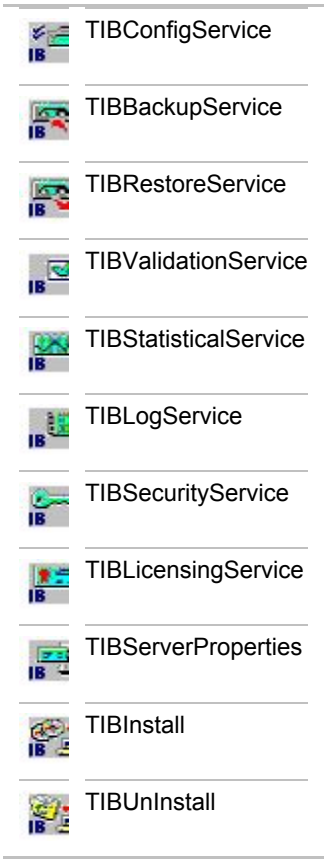
3 Enter events in the Events property string list editor, for example: `IBEvents.Events.Add ('EVENT_NAME');` (for Delphi) or `IBEvents->Events->Add("EVENT_NAME");` (for C++).

4 4. Set the Registered property to true.

InterBase Administration Components

If you have InterBase 6 installed, you can use the InterBase 6 Administration components, which allow you to use access the powerful InterBase Services API calls.

The components are located on the InterBase Admin tab of the IDE and include:



Note: You must install InterBase 6 to use these features.

IBConfigService

Use an TIBConfigService object to configure database parameters, including page buffers, async mode, reserve space, and sweep interval.

IBBackupService

Use an TIBBackupService object to back up your database. With IBBackupService, you can set such parameters as the blocking factor, backup file name, and database backup options.

IBRestoreService

Use an TIBRestoreService object to restore your database. With IBRestoreService, you can set such options as page buffers, page size, and database restore options.

IBValidationService

Use an TIBValidationService object to validate your database and reconcile your database transactions. With the IBValidationService, you can set the default transaction action, return limbo transaction information, and set other database validation options.

IBStatisticalService

Use an TIBStatisticalService object to view database statistics, such as data pages, database log, header pages, index pages, and system relations.

IBLogService

Use an TIBLogService object to create a log file.

IBSecurityService

Use an TIBSecurityService object to manage user access to the InterBase server. With the IBSecurityService, you can create, delete, and modify user accounts, display all users, and set up work groups using SQL roles.

IBLicensingService

Use an TIBLicensingService component to add or remove InterBase software activation certificates.

IBServerProperties

Use an TIBServerProperties component to return database server information, including configuration parameters, and version and license information.

IBInstall

Use an TIBInstall component to set up an InterBase installation component, including the installation source and destination directories, and the components to be installed.

IBUnInstall

Use an TIBUnInstall component to set up an uninstall component.

Deploying Database Applications for the .NET Framework

When deploying database applications using Developer Studio 2006, copy the necessary runtime assemblies and driver DLLs for deployment to a specified location. The following sections list the name of the assemblies and DLLs and the location of where each should reside.

BDP.NET Application Deployment

Copy specific database runtime assemblies to the following location:

Managed Assemblies	Data Provider	Location
Borland.Data.Common.dll	All	GAC
Borland.Data.Provider.dll	All	GAC
Borland.Data.DB2.dll	DB2	GAC
Borland.Data.Interbase.dll	Interbase	GAC
Borland.Data.Mssql.dll	MS SQL/MSDE	GAC
Borland.Data.Oracle.dll	Oracle	GAC
Borland.Data.Msacc.dll	MS Access	GAC
Borland.Data.Sybase.dll	Sybase	GAC

Note: If you are deploying a distributed database application that uses the BDP.NET Remoting components, such as DataHub, DataSync, RemoteConnection, and RemoteServer, you must install Borland.Data.DataSync.dll to the GAC.

Copy unmanaged database driver DLLs to the following location:

DLLs	Data Provider	Location
bdpint20.dll	Interbase	search path
bdpdb220.dll	DB2	search path
bdpmss20.dll	MS SQL/MSDE	search path
bdpora20.dll	Oracle	search path
bdpmsa20.dll	MS Access	search path
bdpsyb20.dll	Sybase	search path

dbExpress for .NET Application Deployment

Copy specific database runtime assemblies to the following location:

Managed Assemblies	Data Provider	Location
Borland.VclDbExpress.dll	All	GAC
Borland.VclDbCtrls.dll	All	GAC
Borland.VclDbxCds.dll	Required by database applications that use client datasets	GAC

You can deploy associated dbExpress.NET drivers and DataSnap DLLs with your executable. Copy unmanaged database driver DLLs to the following location:

DLLs	Data Provider	Location
dbexpinf.dll	Informix	search path
dbexpint.dll	InterBase	search path
dbexpora.dll	Oracle	search path
dbexpdb2.dll	DB2	search path
dbexpmss.dll	MS SQL	search path
dbexpmysql.dll	MySQL 3.23.x	search path
Midas.dll	Required by database applications that use client datasets	search path

dbGo for .NET Application Deployment

There is no need to deploy runtime assemblies or database drivers for dbGo components used in VCL.NET applications. Microsoft Data Access Components (MDAC) version 2.1 or later is required to run applications with dbGo components outside of the IDE. This applies to Win32 VCL applications, as well as VCL.NET applications. Developer Studio 2006 supports MDAC 2.8.

BDE for .NET Application Deployment

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broader range of support for database manipulation. Although you can use the API of the BDE directly in your application, the components on the **BDE** section of the **Tool Palette** wrap most of this functionality for you.

Building Applications with Unmanaged Code

Borland's Developer Studio 2006 provides the capability to work with the .NET features that support unmanaged code. If you have COM or ActiveX components that you want to use within the .NET Framework, you can use the .NET COM Interop capabilities from within Developer Studio 2006 while building your applications. If you have existing CORBA applications or want to build new CORBA applications, you can use the Borland Janeva product from within Developer Studio 2006.

In This Section

Using COM Interop in Managed Applications
Conceptual overview of COM Interop technologies and tools.

Using Platform Invoke with Developer Studio 2006
Describes working with the Platform Invoke services from Developer Studio 2006

Using DrInterop
Describes the drinterop command line tool.

Using COM Interop in Managed Applications

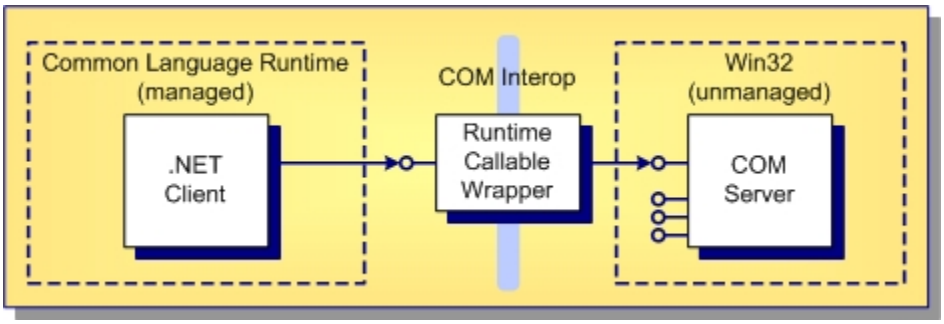
COM Interop is a .NET service that allows seamless interoperation between managed and unmanaged code. The COM Interop service is a two-way bridge: It allows you to leverage existing COM servers and ActiveX Controls in new .NET applications, as well as to expose .NET components in legacy, unmanaged applications.

The Developer Studio 2006 IDE features tools that will help you integrate your legacy COM servers and ActiveX Controls into managed applications. Within the IDE, you can add references to unmanaged DLLs to your project, and then browse the types contained in them, just as you can with managed assemblies. You can add ActiveX Controls to the **Tool Palette**, and then drop them on your forms as you would with any .NET component.

The following topics are covered in this overview:

- Introduction to the terminology of COM Interop. If you are already familiar with these concepts, you can skip directly to the section on Developer Studio 2006 IDE features and tools for COM/Interop.
- Introduction to some of the .NET Framework SDK tools for working with COM/Interop.
- Using COM Interop Assemblies in the IDE.

COM Interop Overview



Seamless interoperability is achieved through stand-in objects called Runtime Callable Wrappers (RCW). The RCW is a layer of communication between your managed application, and the actual unmanaged COM server.

COM Interop Terminology

The .NET Framework has a rich collection of terms and three-letter acronyms. This section will help you understand the terminology you will encounter when reading other COM Interop literature.

Metadata

In the context of .NET and COM, metadata is a term used to mean type information. In COM, type information can be stored in a variety of ways. For instance, a C++ header file is a language-specific container for type information. A type library is also a container for type information, but being a binary format, type libraries are language neutral. Unlike the COM development model where type libraries are not required, language neutral metadata is mandatory for all .NET assemblies. Every assembly is self-describing; its metadata contains complete type information, including private types and private class members.

Custom Attributes

Developers often tag program entities (such as classes and their methods) with descriptive attributes such as static, private, protected, and public. In the .NET Framework, you can tag any entity, including classes, properties, methods, and even assemblies themselves, with an attribute of your own design and meaning. Custom attributes are

expressed in source code, and are processed by the compiler. At the end of the build process, custom attributes are emitted into the output assembly just like all metadata.

Reflection

A unique characteristic of the .NET Framework is that type information is not lost during the compilation process. Instead, all metadata, including custom attributes, is emitted by the compiler into the final output assembly. Metadata is available at runtime, through .NET Reflection services. The .NET Framework SDK provides a reflection tool called ildasm that allows the developer to open any .NET assembly, and inspect the types declared therein. Such reflection tools often allow programmers to directly view the IL code generated by the compiler. The Developer Studio 2006 IDE contains its own integrated reflection tool, in the form of the meta data explorer tool that appears when you open a .NET assembly.

Global Assembly Cache

In COM, components can be deployed anywhere on the user's machine. Usually, a component's installation script records its location in the system registry. Command-line tools such as regsvr32 and tregsvr can also add and remove COM components from the registry. Registration of components is required in COM programming, even if the components are not intended to be shared by multiple applications.

The .NET programming model drastically simplifies deployment of applications and components. On the .NET platform, non-shared components are deployed directly into the application's local installation directory; no registration is required. Alternatively, a non-shared component can be deployed in a directory specified in the application's configuration file. Again, registration is not required for this deployment scenario.

Shared components are installed into a special location called the Global Assembly Cache (GAC). The GAC is an evolution of the system registry (though it is a completely separate mechanism and is not associated with the registry at all). The GAC exists in the file system in a folder called \Windows\Assembly. The .NET Framework supports simultaneous, or "side-by-side" deployment of different versions of the same component. When you view the Global Assembly Cache folder using Windows Explorer, you are actually looking at the GAC through a special shell extension. The shell extension presents all of the assemblies that have been installed into the GAC, with their version, culture, and public key information.

There are three ways to install a .NET component into the GAC. The first way is to use the Framework SDK command-line tool called gacutil, which is discussed below. Another way is to install a component into the GAC is to navigate to the \Windows\Assembly folder using Windows Explorer, and then simply drag and drop the assembly into the directory listing pane. Finally, you can also use the .NET Configuration management tool, which is accessible through the Windows Control Panel.

Strong Names

The concept of a strong name is similar to that of the 128-bit Globally Unique Identifier (GUID) in COM programming. A GUID is a name that is guaranteed to be globally unique. Every .NET assembly has a basic name, which consists of a text string, a version number, and optional culture information. For shared assemblies installed into the GAC, the basic name alone is not enough to guarantee the assembly is uniquely identified. To generate a globally unique name, an encryption key with public and private components is used to generate a digital signature. The signature is then applied to the assembly using the .NET Framework SDK Assembly Linker (al.exe), or by using assembly attributes in source code.

Runtime Callable Wrappers and COM Callable Wrappers

Accessing a component, be it a .NET component or a COM server, is largely transparent. That is, when you are using a COM server in a .NET application, the COM server looks like any other .NET component. Similarly a .NET component, when exposed to an unmanaged application through COM Interop, looks like a COM server. This transparency is accomplished by behind-the-scenes proxies, or wrapper objects.

When you use a COM object in a managed application, the Common Language Runtime (CLR) creates an RCW, which is the interface between managed and unmanaged code. The complexities of data marshaling and reference counting are handled by the RCW. In fact the RCW does not even expose the *IUnknown* and *IDispatch* interfaces.

When you use a .NET component in an unmanaged application, the system creates a stand-in called a COM Callable Wrapper (CCW).

Primary Interop Assembly

In the COM programming model, once a GUID is assigned to a type, the GUID always refers to that specific type no matter where the type appears. For example, a common interface might be defined in many different type libraries, but each separate type library would have to define the interface with the same GUID, so the duplication is not a problem. However, if you generate COM Interop assemblies for these separate type libraries, a new and distinct assembly would be created for each type library. Each of these separate assemblies would contain distinct types (as far as the CLR is concerned). The strong identity and self-describing nature of .NET assemblies is actually working against you in this case. Here, it is leading to a GAC that is cluttered with interop assemblies that all contain RCWs for the same type library. Worse yet, to the CLR each assembly contains distinct and incompatible types, because each one has a different strong name.

To avoid this proliferation of assemblies and potential type incompatibilities, the framework gives you the ability to designate one assembly as the primary interop assembly for a type library. A primary interop assembly is always signed with a strong name, by the original publisher of the type library.

COM Interop Tools in the .NET Framework SDK

Some of the functionality provided by the .NET Framework SDK tools is exposed in the development environment. This section is not intended to be a complete reference for these tools; it is merely a starting point for more exploration of the .NET Framework SDK, and hopefully will give you a bit more understanding of how to work with COM Interop technology in the IDE.

Importing and Exporting Type Libraries

Tlbimp is a command-line tool that you can use to generate a .NET assembly from a type library. Tlbimp will operate on a type library directly, or on an unmanaged DLL that contains a type library as an embedded resource. Note the assembly produced by tlbimp contains code for only the RCW, not for the original COM object itself. Therefore you must still deploy and register the COM object on the end-user's machine. The assembly also contains the types described in the type library, expressed as metadata. Tlbimp uses a command line switch to produce a primary interop assembly.

The .NET Framework SDK contains another command-line tool called tlbexp that is used to create a type library from a .NET assembly. Such an exported type library would then be used to expose the .NET component as a COM server, for use within an unmanaged application.

Importing ActiveX Control Libraries

Aximp is a command-line tool that generates an ActiveX Control wrapper assembly. This assembly is required so that the ActiveX Control can be used on a Windows Form. A special utility is required, because a Windows Form can only host controls that are derived from the System.Windows.Forms.Control class, and the tlbimp utility does not create a wrapper derived from that class.

The aximp tool will generate both interop assemblies (as with tlbimp, this includes dependent assemblies), and the ActiveX wrapper assembly. Like tlbimp, aximp has command-line switches to sign the assemblies produced with a strong name. Unlike tlbimp, aximp cannot generate a primary interop assembly.

Generating Strong Names

If you are deploying a .NET component into the GAC, you will need to sign your assembly with a strong name key. This is done by using a .NET Framework SDK command-line tool called sn. The assembly is signed with the strong name in one of three ways:

- By specifying the strong name key file in the assembly linker (al) command line
- By tagging the assembly with the AssemblyKeyFile attribute
- By using a technique called "delay signing"

When using delay signing, the assembly is signed with the public portion of the key file at build time. Before shipping the assembly, the sn tool is used again to sign the assembly with the private key.

Deploying a .NET Component to the Global Assembly Cache

The .NET Framework SDK utility called gacutil is a command-line program that is used to install, remove, and view components in the GAC. The gacutil command is usable from installation scripts as well as from batch files. The gacutil command supports installation and removal of shared assemblies, with and without the use of reference counting. It is recommended that the non-reference counted command switches be used only during development. Installation scripts that use gacutil to install shared components should always use the reference counted command line switches.

Using COM Interop Assemblies in the IDE

All of the functionality encompassed by the .NET Framework SDK command-line tools is in fact exposed by the .NET Framework Class Library itself. The Developer Studio 2006 IDE also takes advantage of the .NET Framework classes to expose interoperability features. The IDE goes beyond the capabilities of the command-line tools, however, making interoperation with unmanaged components even easier.

Type Libraries and Interop Assemblies

The IDE initiates the creation of interop assemblies through the **Project Manager**. When you add a reference to a DLL to your project, you can select from registered type libraries and unmanaged DLLs, or you can browse to an unregistered component.

The IDE creates one interop assembly for each imported type library or DLL. The assemblies are named Interop. LibraryName.dll, where *LibraryName* is the name of the type library. The name of the library is specified in the library statement in IDL source code, so the file name of the generated assembly might be different from that of the original DLL or type library. Each interop assembly (and all of its dependent assemblies) are added to your project as referenced assemblies. The types contained in the interop assembly are added to a namespace with the same name as the type library; Again, this is derived from the library statement in IDL source code.

If the assembly you reference has a primary interop assembly, the IDE will recognize this and avoid generating a new interop assembly. In this case, the IDE will add a reference to the primary interop assembly in the GAC, and it will not copy the assembly to your local project directory.

Importing ActiveX Controls

To use an ActiveX Control in your managed application, you must first add the control to the tool palette. This will create both an interop assembly, and an ActiveX assembly with a wrapper class derived from System.Windows. Forms.AxHost. The ActiveX wrapper assembly will be named AxInterop.LibraryName.dll, where *LibraryName* is the name of the type library. Dragging the control from the palette onto a Windows Form will automatically add references to both assemblies to your project.

Once on your form, the ActiveX Control can be treated as any other .NET component. You can select the control, and set its properties and event handlers in the **Object Inspector**. The ActiveX Control wrapper will expose the

properties of the Windows.Forms.Control class, while properties exposed by the ActiveX Control will be grouped under the *Misc* category.

Interop Assemblies and the Project Manager

Interop assemblies (including ActiveX Control wrapper assemblies) generated by the IDE are kept in a separate folder called COMImports, underneath your project. Each generated assembly will have its 'Copy Local' property set, meaning that when the project is built, the assembly will be copied to the folder where the final build target of the project is kept. The exceptions to this rule are primary interop assemblies, which are deployed in the GAC. When you add a reference to a primary interop assembly, the IDE will not copy the assembly to the COMImports folder. The assembly will still be shown in the **Project Manager**, however, if you right click on it to display its properties, you will notice that the 'Copy Local' setting is turned off.

The list of referenced assemblies (including those that are not interop assemblies) is an attribute of your project. If the COMImports folder (or one of the interop assemblies contained therein) does not exist when you open a project, the IDE will attempt to recreate it. If the IDE cannot create an interop assembly, it will still be shown as a referenced assembly in the **Project Manager**; the IDE will highlight such an assembly so that you know it currently does not exist (or is not registered) on the machine.

Using Platform Invoke with Developer Studio 2006

This topic describes the basic techniques of using unmanaged APIs from Developer Studio 2006. Some of the common mistakes and pitfalls are pointed out, and a quick reference for translating Delphi data types is provided. This topic does not attempt to explain the basics of platform invoke or marshaling data. Please refer to the links at the end of this topic for more information on platform invoke and marshaling. Understanding attributes and how they are used is also highly recommended before reading this document.

The Win32 API is used for several examples. For further details on the API functions mentioned, please see the Windows Platform SDK documentation.

The following topics are discussed in this section:

- Calling unmanaged functions
- Structures
- Callback functions
- Passing Object References
- Using COM Interfaces

Calling Unmanaged Functions

When calling unmanaged functions, a managed declaration of the function must be created that represents the unmanaged types. In many cases functions take pointers to data that can be of variable types. One example of such a function is the Win32 API function `SystemParametersInfo` that is declared as follows:

```
BOOL SystemParametersInfo(
  UINT uiAction,  // system parameter to retrieve or set
  UINT uiParam,   // depends on action to be taken
  PVOID pvParam,  // depends on action to be taken
  UINT fWinIni    // user profile update option
);
```

Depending on the value of `uiAction`, `pvParam` can be one of dozens of different structures or simple data types. Since there is no way to represent this with one single managed declaration, multiple overloaded versions of the function must be declared (see Borland.Vcl.Windows.pas), where each overload covers one specific case. The parameter `pvParam` can also be given the generic declaration IntPtr. This places the burden of marshaling on the caller, rather than the built in marshaler. Note that the data types used in a managed declaration of an unmanaged function must be types that the default marshaler supports. Otherwise, the caller must declare the parameter as IntPtr and be responsible for marshaling the data.

Data Types

Most data types do not need to be changed, except for pointer and string types. The following table shows commonly used data types, and how to translate them for managed code:

Unmanaged Data Type	Managed Data Type	
	Input Parameter	Output Parameter
Pointer to string (PChar)	String	StringBuilder
Untyped parameter/buffer	TBytes	TBytes
Pointer to structure (PRect)	const TRect	var TRect

Pointer to simple type (PByte)	const Byte	var Byte
Pointer to array (PInteger)	array of Integer	array of Integer
Pointer to pointer type (^PInteger)	IntPtr	IntPtr

IntPtr can also represent all pointer and string types, in which case you need to manually marshal data using the Marshal class. When working with functions that receive a text buffer, the StringBuilder class provides the easiest solution. The following example shows how to use a StringBuilder to receive a text buffer:

```
function GetText(Window: HWND; BufSize: Integer = 1024): string;
var
  Buffer: StringBuilder;
begin
  Buffer := StringBuilder.Create(BufSize);
  GetWindowText(Window, Buffer, Buffer.Capacity);
  Result := Buffer.ToString;
end;
```

The StringBuilder class is automatically marshaled into an unmanaged buffer and back. In some cases it may not be practical, or possible, to use a StringBuilder. The following examples show how to marshal data to send and retrieve strings using `SendMessage`:

```
procedure SetText(Window: HWND; Text: string);
var
  Buffer: IntPtr;
begin
  Buffer := Marshal.StringToHGlobalAuto(Text);
  try
    Result := SendMessage(Window, WM_SETTEXT, 0, Buffer);
  finally
    Marshal.FreeHGlobal(Buffer);
  end;
end;
```

An unmanaged buffer is allocated, and the string copied into it by calling `StringToHGlobalAuto`. The buffer must be freed once it's no longer needed. To marshal a pointer to a structure, use the Marshal. StructureToPtr method to copy the contents of the structure into the unmanaged memory buffer.

The following example shows how to receive a text buffer and marshal the data into a string:

```
function GetText(Window: HWND; BufSize: Integer = 1024): string;
var
  Buffer: IntPtr;
begin
  Buffer := Marshal.AllocHGlobal(BufSize * Marshal.SystemDefaultCharSize);
  try
    SendMessage(Window, WM_GETTEXT, BufSize, Buffer);
    Result := Marshal.PtrToStringAuto(Buffer);
  finally
    Marshal.FreeHGlobal(Buffer);
  end;
end;
```

It is important to ensure the buffer is large enough, and by using the SystemDefaultCharSize method, the buffer is guaranteed to hold `BufSize` characters on any system.

Advanced Techniques

When working with unmanaged API's, it is common to pass parameters as either a pointer to something, or `NULL`. Since the managed API translations don't use pointer types, it might be necessary to create an additional overloaded version of the function with the parameter that can be `NULL` declared as IntPtr.

Special Cases

There are cases where a StringBuilder and even the Marshal class will be unable to correctly handle the data that needs to be passed to an unmanaged function. An example of such a case is when the string you need to pass, or receive, contains multiple strings separated by `NULL` characters. Since the default marshaler will consider the first `NULL` to be the end of the string, the data will be truncated (this also applies to the `StringToHGlobalXXX` and `PtrToStringXXX` methods). In this situation `TBytes` can be used (using the PlatformStringOf and PlatformBytesOf functions in `Borland.Delphi.System` to convert the byte array to/from a string). Note that these utility functions do not add or remove terminating `NULL` characters.

When working with COM interfaces, the UnmanagedType enumeration (used by the MarshalAsAttribute class) has a special value, `LPStruct`. This is only valid in combination with a System.Guid class, causing the marshaler to convert the parameter into a Win32 GUID structure. The function `CoCreateInstance` that is declared in Delphi 7 as:

```
function CoCreateInstance([MarshalAs(UnmanagedType.LPStruct)] clsid: TCLSID;
  [MarshalAs(UnmanagedType.IUnknown)] unkOuter: TObject;
  dwClsContext: Longint;
  [MarshalAs(UnmanagedType.LPStruct)] iid: TIID;
  [MarshalAs(UnmanagedType.Interface)] out pv
): HResult;
```

This is currently the only documented use for `UnmanagedType.LPStruct`.

Structures

The biggest difference between calling unmanaged functions and passing structures to unmanaged functions is that the default marshaler has some major restrictions when working with structures. The most important are that dynamic arrays, arrays of structures and the StringBuilder class cannot be used in structures. For these cases IntPtr is required (although in some cases string paired with various marshaling attributes can be used for strings).

Data Types

The following table shows commonly used data types, and how to "translate" them for managed code:

Unmanaged Data Type	Managed Data Type	
	Input Parameter	Output Parameter
Pointer to string (PChar)	String	IntPtr
Character array (`array[a..b] of Char`)	String	String
Array of value type (`array[a..b] of Byte`)	`array[a..b] of Byte`	`array[a..b] of Byte`
Dynamic array (`array[0..0] of type`)	IntPtr	IntPtr

Array of struct (`array[1..2] of TRect`)	IntPtr or flatten	IntPtr or flatten
Pointer to structure (PRect)	IntPtr	IntPtr
Pointer to simple type (PByte)	IntPtr	IntPtr
Pointer to array (PInteger)	IntPtr	IntPtr
Pointer to pointer type (^PInteger)	IntPtr	IntPtr

When working with arrays and strings in structures, the MarshalAs attribute is used to describe additional information to the default marshaler about the data type. A record declared in Delphi 7, for example:

```
type
  TMyRecord = record
    IntBuffer: array[0..31] of Integer;
    CharBuffer: array[0..127] of Char;
    lpszInput: LPTSTR;
    lpszOutput: LPTSTR;
  end;
```

Would be declared as follows in Developer Studio 2006:

```
type
  [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
  TMyRecord = record
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 32)]
    IntBuffer: array[0..31] of Integer;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    CharBuffer: string;
    [MarshalAs(UnmanagedType.LPTStr)]
    lpszInput: string;
    lpszOutput: IntPtr;
  end;
```

The above declarations assume that the strings contain platform dependant TChar's (as commonly used by the Win32 API). It is important to note that in order to receive text in `lpszOutput`, the Marshal. AllocHGlobal method needs to be called before passing the structure to an API function.

A structure can contain structures, but not pointers to structures. For such cases an IntPtr must be declared, and the Marshal. StructureToPtr method used to move data from the managed structure into unmanaged memory. Note that StructureToPtr does not allocate the memory needed (this must be done separately). Be sure to use Marshal. SizeOf to determine the amount of memory required, as Delphi's SizeOf is not aware of the MarshalAs attribute (in the example above, `CharBuffer` would be 4 bytes using Delphi's SizeOf when it in fact should occupies 128 bytes on a single byte system). The following examples show how to send messages that pass pointers to a structure:

```
procedure SetRect(Handle: HWND; const Rect: TRect);
var
  Buffer: IntPtr;
begin
  Buffer := Marshal.AllocHGlobal(Marshal.SizeOf(TypeOf(TRect)));
  try
    Marshal.StructureToPtr(TObject(Rect), Buffer, False);
    SendMessage(Handle, EM_SETRECT, 0, Buffer);
  finally
    Marshal.DestroyStructure(Buffer, TypeOf(TRect));
```

```
  end;
end;

procedure GetRect(Handle: HWND; var Rect: TRect);
var
  Buffer: IntPtr;
begin
  Buffer := Marshal.AllocHGlobal(Marshal.SizeOf(TypeOf(TRect)));
  try
    SendMessage(Handle, EM_GETRECT, 0, Buffer);
    Rect := TRect(Marshal.PtrToStructure(Buffer, TypeOf(TRect)));
  finally
    Marshal.DestroyStructure(Buffer, TypeOf(TRect));
  end;
end;
```

It is important to call DestroyStructure rather than FreeHGlobal if the structure contains fields where the marshaling layer needs to free additional buffers (see the documentation for DestroyStructure for more details).

Advanced topics

Working with unmanaged API's it is not uncommon to need to convert a byte array into a structure (or retrieve one or more fields from a structure held in a byte array), or vice versa. Although the Marshal class contains a method to retrieve the offset of a given field, it is extremely slow and should be avoided in most situations. Informal performance tests show that for a structure with eight or nine numeric fields, it is much faster to allocate a block of unmanaged memory, copy the byte array to the unmanaged memory and call PtrToStructure than finding the position of just one field using Marshal. OffsetOf and converting the data using the BitConverter class. `Borland.Vcl.WinUtils` contains helper functions to perform conversions between byte arrays and structures (see StructureToBytes and BytesToStructure).

Special cases

There are cases where custom processing is required, such as sending a message with a pointer to an array of integers. For situations like this, the Marshal class provides methods to copy data directly to the unmanaged buffer, at specified offsets (so you can construct an array of a custom data type after allocating a buffer). The following example shows how to send a message where the `LParam` is a pointer to an array of Integer:

```
function SendArrayMessage(Handle: HWND; Msg: UINT; WParam: WPARAM;
  LParam: TIntegerDynArray): LRESULT;
var
  Buffer: IntPtr;
begin
  Buffer := Marshal.AllocHGlobal(Length(LParam) * SizeOf(Integer));
  try
    Marshal.Copy(LParam, 0, Buffer, Length(LParam));
    Result := SendMessage(Handle, Msg, WParam, Buffer);
  finally
    Marshal.FreeHGlobal(Buffer);
  end;
end;
```

Callback Functions

When passing a function pointer for a managed function to an unmanaged API, a reference must be maintained to the delegate or it will be garbage collected. If you pass a pointer to your managed function directly, a temporary

delegate will be created, and as soon as it goes out of scope (at the end of `MyFunction` in the example below), it is subject to garbage collection. Consider the following Delphi 7 code:

```
function MyFunction: Integer;
begin
  ...
  RegisterCallback(@MyCallback);
  ...
end;
```

In order for this to work in a managed environment, the code needs to be changed to the following:

```
const
  MyCallbackDelegate: TFNMyCallback = @MyCallback;

function MyFunction: Integer;
begin
  ...
  RegisterCallback(MyCallbackDelegate);
  ...
end;
```

This will ensure that the callback can be called as long as `MyCallbackDelegate` is in scope.

Data types

The same rules apply for callbacks as any other unmanaged API function.

Special cases

Any parameters used in an asynchronous process must be declared as IntPtr. The marshaler will free any memory it has allocated for unmanaged types when it returns from the function call. When using an IntPtr, it is your responsibility to free any memory that has been allocated.

Passing Object References

When working with for example the Windows API, object references are sometimes passed to the API where they are stored and later passed back to the application for processing usually associated with a given event. This can still be accomplished in .NET, but special care needs to be taken to ensure a reference is kept to all objects (otherwise they can and will be garbage collected).

Data types

Unmanaged Data Types	Managed Data Type	
	Supply Data	Receive Data
Pointer (Object reference, user data)	GCHandle	GCHandle

The GCHandle provides the primary means of passing an object references to unmanaged code, and ensuring garbage collection does not happen. A GCHandle needs to be allocated, and later freed when no longer needed. There are several types of GCHandle, GCHandleType.Normal being the most useful when an unmanaged client holds the only reference. In order pass a GCHandle to an API function once it is allocated, type cast it to IntPtr (and

optionally onwards to LongInt, depending on the unmanaged declaration). The IntPtr can later be cast back to a GCHandle. Note that IsAllocated must be called before accessing the Target property, as shown below:

```
procedure MyProcedure;
var
  Ptr: IntPtr;
  Handle: GCHandle;
begin
  ...
  if Ptr <> nil then
  begin
    Handle := GCHandle(Ptr);
    if Handle.IsAllocated then
      DoSomething(Handle.Target);
  end;
  ...
end;
```

Advanced techniques

The use of a GCHandle, although relatively easy, is fairly expensive in terms of performance. It also has the possibility of resource leaks if handles aren't freed correctly. If object references are maintained in the managed code, it is possible to pass a unique index, for example the hash code returned by the GetHashCode method, to the unmanaged API instead of an object reference. A hash table can be maintained on the managed side to facilitate retrieving an object instance from a hash value if needed. An example of using this technique can be found in the TTreeNodes class (in `Borland.Vcl.ComCtrls`).

Using COM Interfaces

When using COM interfaces, a similar approach is taken as when using unmanaged API's. The interface needs to be declared, using custom attributes to describe the type interface and the GUID. Next the methods are declared; using the same approach as for unmanaged API's. The following example uses the IAutoComplete interface, defined as follows in Delphi 7:

```
IAutoComplete = interface(IUnknown)
  ['{00bb2762-6a77-11d0-a535-00c04fd7d062}']
  function Init(hwndEdit: HWND; punkACL: IUnknown;
    pwszRegKeyPath: LPCWSTR; pwszQuickComplete: LPCWSTR): HRESULT; stdcall;
  function Enable(fEnable: BOOL): HRESULT; stdcall;
end;
```

In Developer Studio 2006 it is declared as follows:

```
[ComImport, GuidAttribute('00BB2762-6A77-11D0-A535-00C04FD7D062'), InterfaceTypeAttribute
(ComInterfaceType.InterfaceIsIUnknown)]
IAutoComplete = interface
  function Init(hwndEdit: HWND; punkACL: IEnumString;
    pwszRegKeyPath: IntPtr; pwszQuickComplete: IntPtr): HRESULT;
  function Enable(fEnable: BOOL): HRESULT;
end;
```

Note the custom attributes used to describe the GUID and type of interface. It is also essential to use the ComImportAttribute class. There are some important notes when importing COM interfaces. You do not need to implement the `IUnknown`/`IDispatch` methods, and inheritance is not supported.

Data types

The same rules as unmanaged functions apply for most data types, with the following additions:

Unmanaged Data Type	Managed Data Type	
	Supply Data	Receive Data
GUID	System.Guid	System.Guid
IUnknown	TObject	TObject
IDispatch	TObject	TObject
Interface	TObject	TObject
Variant	TObject	TObject
SafeArray (of type)	array of <type>	array of <type>
BSTR	String	String

Using the MarshalAsAttribute custom attribute is required for some of the above uses of TObject, specifying the exact unmanaged type (such as UnmanagedType.IUnknown, UnmanagedType.IDispatch or UnmanagedType.Interface). This is also true for certain array types. An example of explicitly specifying the unmanaged type is the `Next` method of the `IEnumString` interface. The Win32 API declares `Next` as follows:

```
HRESULT Next(
  ULONG celt,
  LPOLESTR * rgelt,
  ULONG * pceltFetched
);
```

In Developer Studio 2006 the declaration would be:

```
function Next(celt: Longint;
  [out, MarshalAs(UnmanagedType.LPArray, ArraySubType = UnmanagedType.LPWStr,
SizeParamIndex = 0)]
  rgelt: array of string;
  out pceltFetched: Longint
): Integer;
```

Advanced techniques

When working with safearrays, the marshal layer automatically converts (for example) an array of bytes into the corresponding safearray type. The marshal layer is very sensitive to type mismatches when converting safearrays. If the type of the safearray does not exactly match the type of the managed array, an exception is thrown. Some of the Win32 safearray API's do not set the type of the safearray correctly when the array is created, which will lead to a type mismatch in the marshal layer when used from .NET. The solutions are to either ensure that the safearray is created correctly, or to bypass the marshal layer's automatic conversion. The latter choice may be risky (but could be the only alternative if you don't have the ability to change the COM server that is providing the data). Consider the following declaration:

```
function AS_GetRecords(const ProviderName: WideString; Count: Integer;
  out RecsOut: Integer; Options: Integer; const CommandText: WideString;
  var Params: OleVariant; var OwnerData: OleVariant): OleVariant;
```

If the return value is known to always be a safearray (that doesn't describe its type correctly) wrapped in a variant, we can change the declaration to the following:

```
type
  TSafeByteArrayData = packed record
    VType: Word;
    Reserved1: Word;
    Reserved2: Word;
    Reserved3: Word;
    VArray: IntPtr;  { This is a pointer to the actual SafeArray }
  end;

function AS_GetRecords(const ProviderName: WideString; Count: Integer;
  out RecsOut: Integer; Options: Integer; const CommandText: WideString;
  var Params: OleVariant; var OwnerData: OleVariant): TSafeByteArrayData;
```

Knowing that an OleVariant is a record, the TSafeByteArrayData record can be extracted from Delphi 7's TVarData (equivalent to the case where the data type is varArray). The record will provide access to the raw pointer to the safearray, from which data can be extracted. By using a structure instead of an OleVariant, the marshal layer will not try to interpret the type of data in the array. You will however be burdened with extracting the data from the actual safearray.

Special cases

Although it is preferred to use Activator.CreateInstance when creating an instance, it is not fully compatible with `CoCreateInstanceEx`. When working with remote servers, CreateInstance will always try to invoke the server locally, before attempting to invoke the server on the remote machine. Currently the only known work-around is to use CoCreateInstanceEx.

Since inheritance isn't supported, a descendant interface needs to declare the ancestor's methods. Below is the IAutoComplete2 interface, which extends IAutoComplete.

```
[ComImport, GuidAttribute('EAC04BC0-3791-11d2-BB95-0060977B464C'), InterfaceTypeAttribute
(ComInterfaceType.InterfaceIsIUnknown)]
IAutoComplete2 = interface(IAutoComplete)
  // IAutoComplete methods
  function Init(hwndEdit: HWND; punkACL: IEnumString;
    pwszRegKeyPath: IntPtr; pwszQuickComplete: IntPtr): HRESULT;
  function Enable(fEnable: BOOL): HRESULT;
  //
  function SetOptions(dwFlag: DWORD): HRESULT;
  function GetOptions(var dwFlag: DWORD): HRESULT;
end;
```

Virtual Library Interfaces

This topic describes how to use a feature of Delphi called Virtual Library Interfaces. Virtual Library Interfaces allows you to discover, load, and call unmanaged code at runtime, without the use of the `DllImport` attribute.

Standard PInvoke

To call an unmanaged function from managed code, you must use a .NET service called Platform Invoke, or PInvoke. The Platform Invoke service requires you to declare in source code, a prototype for each unmanaged function you wish to call. You can do this either within an existing .NET class, or you can create an entirely new class to organize the prototypes. You must also tag each unnamaged prototype declaration with the `DllImport` attribute.

The `DllImport` attribute requires you to specify the name of the DLL in which the unmanaged function resides. Since the unmanaged prototype is tagged with the `DllImport` attribute at compile-time, dynamic discovery of DLLs and their exported unmanaged functions is difficult. Furthermore, if the unmanaged function is not actually exported from the DLL named in the `DllImport` attribute, a runtime failure will result. To avoid a runtime failure, you would have to use `LoadLibrary` to load the exact DLL you require, and then call `GetProcAddress` to verify the existance of the unmanaged function. Even so, you would not be able to directly call the function using the pointer returned from `GetProcAddress`. Instead you would have to pass the pointer along to a function in another unmanaged DLL. That function would then use the pointer to make the call.

Using Virtual Library Interfaces

Virtual Library Interfaces still must use the Platform Invoke service to call unmanaged code. However, instead of using the `DllImport` attribute, Virtual Library Interfaces creates an interface on the unmanaged DLL at runtime, using methods of the .NET `System.Reflection.Emit` namespace.

Using Virtual Library Interfaces requires that you do three things:

- Add `Borland.Vcl.Win32` to the uses clause.
- Declare an interface containing the exported, unmanaged functions you wish to call.
- Call the Supports function to ensure that the unmanaged DLL exists and that the functions in the interface declaration are actually exported.

If the Supports function returns True, then the DLL supports all of the functions named in the interface declaration, so you know it is safe to call them. Within the interface declaration, you do not need to use the `DllImport` attribute on the prototypes.

For example, if you have a DLL called MyFunctions.dll, that contains the following exported functions:

```
function AFunction       : Boolean;
function AnotherFunction : Boolean;
```

To call these functions from managed code, add the `Borland.Vcl.Win32` unit to the uses clause and declare an interface in Delphi:

```
uses Borland.Vcl.Win32, ...;
...
type
IMyFunctions = interface
['Your GUID'] // Not strictly required, but good practice
function AFunction       : Boolean;
```

```
function AnotherFunction : Boolean;
end;
```

The signature of the Supports function is:

```
function Supports(ModuleName: string; Source: System.Type; var Instance) : Boolean;
```

To call the unmanaged functions, first call Supports to load the DLL, and create the interface on the DLL:

```
var
MyFunctions : IMyFunctions;
begin
  if Supports("MyFunctions.dll", IMyFunctions, MyFunctions) then
     if MyFunctions.AFunction then
      begin
        ...
      end;
  end;
end;
```

Virtual Library Interfaces have the same limitations in terms of compatible native parameter types and their mapping to .NET types. In addition, all unmanaged functions are expected to use the stdcall calling convention.

Using DrInterop

The drinterop command line tool examines an assembly and produces a set of diagnostic messages that help you prepare the assembly for use with COM/Interop.

The drinterop tool is located in the bin directory of the product installation. It is invoked by typing

```
drinterop assembly
```

Message	Cause
Assembly `ComVisible` attribute is `true` when it should be `false`.	The `[assembly:ComVisible(bool)]` attribute is set to `true`, or is not present. Assemblies should be hidden from COM to reduce registry clutter. Set the `ComVisible` attribute to `false`, and selectively expose classes and interfaces.
Assembly, class, or interface is exposed to COM but does not contain the `Guid` attribute.	The assembly, class, or interface has the `ComVisible` attribute set to `true` but does not contain a `Guid` attribute.
A type library should be generated and registered for assembly.	This message is generated when a type library is not found in the same directory as the assembly.
Assembly does not contain the `TypeLibVersion` attribute.	The assembly does not contain the `[assembly: TypeLibVersion(x,y)]` attribute. By default type library version numbers are generated using only the first two numbers of the assembly version. Using the TypeLibVersion attribute can help avoid problems where two assemblies would produce the same type library because the first two digits of their version number are the same.
Reduce registry size by adding attribute `[ClassInterface(ClassInterfaceType.None)]` to class.	The class does not contain the `ClassInterface` attribute. By default, each class will cause the creation of a corresponding interface with the class name prefixed with an underscore character. This interface has no methods associated with it. You can reduce registry size and clutter by putting the `[ClassInterface(ClassInterfaceType.None)]` attribute on the class.

Note: The drinterop tool will not print any messages if it does not find any of the above conditions.

Deploying COM Interop Applications

Two things are important to keep in mind when working with unmanaged components. First, remember that an interop assembly is not a replacement for the COM server; it is a stand-in, or proxy for it. The interop assemblies produced by tlbimp and Developer Studio 2006 are not transformations of the component's unmanaged code into managed code. Every file required by the component in an unmanaged deployment environment, must also be deployed in a managed environment *in addition to* the interop assemblies. Second, the .NET Framework's interop services do not circumvent the requirement of registering the COM server on the end-user's machine. Note the registration requirement also applies during the development of your managed application.

As with any other .NET assembly, an interop assembly can be deployed alongside the managed executable in the installation folder, or it can be deployed in the GAC. If you deploy the interop assembly into the GAC, you must give it a strong name during development. Primary interop assemblies are always deployed into the GAC; however, just because an assembly is deployed to the GAC, does not automatically make it a primary interop assembly. An interop assembly is designated as a primary interop assembly by using the /primary command-line option of the tlbimp utility. The IDE currently has no built-in support for creating primary interop assemblies. Unmanaged COM servers can be deployed anywhere on the end-user's machine, however, as noted previously, you must still register unmanaged components when your application is installed.

Building Reports for .NET Applications

Developer Studio 2006 ships with Rave Reports from Nevrona. Using the report components, you can build full-featured reports for your applications. You can create solutions that include reporting capabilities which can be used and customized by your customers. Additionally, the ComponentOne tools that ship with Developer Studio 2006 include components for creating and generating reports.

In This Section

Using Rave Reports in Developer Studio 2006
Describes how Developer Studio 2006 supports integration of Rave Reports objects.

Using Rave Reports in Developer Studio 2006

The Developer Studio 2006 environment supports the integration of report objects in your applications. This integration allows you to create a report using the Rave Reports Designer or to add Rave Reports ActiveX components directly onto your Windows Forms and Web Forms in the Developer Studio 2006**Designer**. Your application users can create and display their own reports, or display existing reports. The Developer Studio 2006 integration with Rave Reports allows you to:

- Include new report objects in projects.
- Add Rave Reports ActiveX objects onto Windows Forms and Web Forms.

Creating New Reports in Developer Studio 2006

You can include Rave reports in Developer Studio 2006 just as you would other third-party components. The report is stored as a separate Rave Report object. You can reference the report in other applications that need to call or generate that report. When you create a new application, you can include the report object by adding a reference to it in the **Project Manager**. Rave Reports also provide the capability to connect your report object to a datasource, which allows your application to build the report dynamically, based on current database information.

Using Rave Reports ActiveX Components

You can add any Rave Reports ActiveX objects to your applications. The Developer Studio 2006**Tool Palette** provides a list of any available ActiveX objects. Just drag the objects you want onto a Windows Form or a Web Form during design. Fill in the appropriate properties and modify any code in the **Code Editor**. You may need to reset your .NET components and select the ActiveX components from the **Installed .NET Components** dialog.

Procedures

ASP.NET Procedures

Adding Aggregate Values with DBWebAggregateControl

You can use DBWebAggregateControl to apply one of several standard aggregation functions to a data column. The control displays the aggregate value in a text box, which also support a linked caption.

To create and configure a DBWebAggregateControl

1. Create a new ASP.NET web application and add your database connection, data adapter, dataset, and DBWebDataSource component to the application..
2. Set the Active property of BdpDataAdapter to **True**.
3. Place a DBWebAggregateControl component on the **Web Form Designer**.
4. Set the DBDataSource property of the DBWebAggregateControl to your DBWebDataSource1, which is the default name of the DBWebDataSource component.
5. Set the TableName property.
6. Choose the AggregateType property value from the drop down list.
7. Choose the ColumnName property from the drop down list.

   The text box is filled with the value based on the type of aggregate you selected and the values in the column you selected.

> **Note:** If you think there may be NULL values in your selected column, set the IgnoreNullValues property to **True**, otherwise you may get an error.

To set the caption for DBWebAggregateControl

1. In the **Object Inspector** enter the caption in the Caption property field.
2. Choose a position from the CaptionPosition property drop down list.

Adding Web References in ASP.NET Projects

If you want to consume a web service, you must create a client application, and add a Web Reference. These procedures describe how to create an ASP.NET client application that consumes a third-party web service. The client application consumes the DeadOrAliveWS web service available from the XMethods Web site. This web service lets you query a simple database of celebrities and their respective birthdates and expiration dates.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box appears.

2. Double-click the **ASP.NET Web Application** icon in either the **C# Projects** or **Delphi for .NET Projects** item categories.

   The **New ASP.NET Application** dialog box appears.

3. In the **Name** field, enter a name for your project.
4. In the **Location** field, enter a path for your project.

   Tip: Most ASP.NET projects reside in the IIS directory Inetpub\wwwroot.

5. If necessary, click the **View Server Options** button to change your Web server settings.

   Tip: The default Server Options will usually be sufficient, so this step is optional.

6. Click **OK**.

   The Web Forms Designer appears.

To design the ASP.NET web page

1. If necessary, click **Design** view.
2. From the **Web Controls** category of the **Tool Palette**, place a Button component onto the Designer surface.

   The Button control appears on the Designer. Make sure the control is selected.

3. In **Object Inspector**, set the **Text** property to Dead or Alive?.
4. From the **Web Controls** category of the **Tool Palette**, place a TextBox component onto the Designer above the Button.

   This is where you type your query to the Web Service.

5. Place a Label component below the Button.

   This is where the results of the web service query are displayed.

Use the UDDI browser to locate the DeadOrAlive Web Service on the internet. This allows you to use the methods and objects published by the Web Service Definition Language (WSDL).

To add the Web Reference for DeadOrAliveWS

1. Choose **Project** ▸ **Add Web Reference**.
2. In the **Borland UDDI Browser** web dialog box, click the **XMethods Full** link in the list of available UDDI directories.

   A list of various web services published on the XMethods Web site appears.

3 Find and click the **DeadOrAliveWS** link.

Tip: You can use Ctrl+F to search within the **Borland UDDI Browser**.

4 Click the link to the WSDL file:

```
http://www.abundanttech.com/webservices/deadoralive/deadoralive.wsdl
```

A WSDL document appears. This XML document describes the interface to the DeadOrAliveWS web service.

5 Click **Add Reference** to add the WSDL document to the client application.

A **Web References** folder containing a **com.abundanttech.www** node is added to the Project directory in the **Project Manager**.

To write the application logic

1 If necessary, click **Design** view.

2 Double-click the **Dead or Alive?** button to view the code-behind file.

3 For a Delphi for .NET Web Services application, implement the Click event in the **Code Editor** with the following code :

```
[Delphi]
procedure TWebForm1.Button1_Click(sender: System.Object; e: System.EventArgs);
var
  result: DataSet;
  ws: DeadOrAlive;
  currentTable: DataTable;
  currentRow: DataRow;
  currentCol: DataColumn;
begin
  //This initializes the web service
  ws := DeadOrAlive.Create;

  //Send input to the web service
  result := ws.getDeadOrAlive(TextBox1.Text);

  //parse results and display them
  Label1.Text := '';
  for currentTable in result.Tables do
    begin
      Label1.Text := Label1.Text + '<p>' + #13#10;
      for currentRow in currentTable.Rows do
        begin
          for currentCol in currentTable.Columns do
            begin
              Label1.Text := Label1.Text + currentCol.ColumnName + ': ';
              Label1.Text := Label1.Text + (currentRow[currentCol]).ToString;
              Label1.Text := Label1.Text  + '<br>' + #13#10;
            end;
        end;
      Label1.Text := Label1.Text + '</p>';
    end;

end;
```

When you added the Web Reference to your application, Developer Studio 2006 used the WSDL to generate a proxy class representing the "Hello World" web service. The Click event uses methods from the proxy class to access the web service. For Delphi for .NET Web Services, you may need to add the unit name of the proxy class, abundanttech.deadoralive, to the uses clause of your Web Form unit to prevent errors in your Click event.

4 For a C# Web Services application, implement the Click event in the **Code Editor** with the following code :

```
[C#]
private void button1_Click(object sender, System.EventArgs e)
{
        DataSet result;

        //This initializes the web service
        DeadOrAlive source = new DeadOrAlive();

        //Send input to the web service
        result = source.getDeadOrAlive(textBox1.Text);

        //parse results and display them
        label1.Text = "";
        foreach (DataTable currentTable in result.Tables) {
                label1.Text += "<p>\n";
                foreach (DataRow currentRow in currentTable.Rows) {
                        foreach (DataColumn currentCol in currentTable.Columns) {
                                label1.Text += currentCol.ColumnName + ": ";
                                label1.Text += currentRow[currentCol] + "<br>\n";
                        }
                }
            label1.Text += "</p>";
        }
}
```

Note: As you can see by the added application logic code, the DeadOrAliveWS web service returns query results in the form of a dataset. Web Services can, however, return data in a variety of formats.

To run the application

1 Choose **Project** ▸ **Build All Projects**.

Now your project is built and resides on your ASP.NET server.

2 Open a Web browser.

3 Type the URL of your Web Application's .aspx file and press Enter.

Tip: If you are using Microsoft IIS, the URL is the path of the .aspx file after Inetpub\wwwroot. For example, if the path of your Web Application is c:\Inetpub\wwwroot\WebApplication1 and your .aspx file is named "WebForm1.aspx", the URL would be http://localhost/WebApplication1/WebForm1.aspx.

4 If necessary, enter your user name and password for your ASP.NET server.

The web page for your web application appears.

5 Enter the name of a celebrity (for example, Isaac Asimov) in the text box and click the **Dead or Alive?** button.

Your web application requests the information from the DeadOrAliveWS web service and displays the result in the label.

Note: If no information is displayed, that name may not be in the database. Check your spelling or try a different name.

Binding Columns in the DBWebGrid

There may be times when you want to modify the order in which columns appear in a DBWebGrid control. You can accomplish this task by binding columns manually, from within the **Property Builder**.

To open the Property Builder

1. Start a new ASP.NET application.
2. Add a data provider.
3. Add a DBWebDataSource object and connect it to a generated dataset.
4. Add a DBWebGrid control to your Web form.
5. Click the **Property Builder Designer verb**, located at the bottom of the **Object Inspector**.

   This displays the **Property Builder**.

To change column order

1. On the **Property Builder**, click the **General** tab.
2. Set the **DataSource** to the DBWebDataSource, or to the dataset the DBWebDataSource points to.
3. Click the **Columns** tab.
4. Select the columns you want to appear in the **Available Columns** list.
5. Click the right-arrow button to add the columns to the **Selected Columns** list.
6. Rearrange the column order, if you like, in the **Selected Columns** list.
7. You can change the column heading name as it appears in the grid by changing the **Header** text.
8. Click **Apply**.
9. Click **OK**.

Warning: If you choose to bind columns in this way, you must set the AutoGenerateColumns property to **False**. Setting this property to **True** raises a runtime error, and does not allow the visible restriction of columns at designtime. If the same column is bound to a grid more than once, you may get a runtime error.

Building an Application with DB Web Controls

The following procedures describe the minimum number of steps required to build a simple ASP.NET database application using DB Web Controls and BDP.NET. After generating the required connection objects, the project displays data in a DBWebGrid with a DBWebNavigator. Additional information is provided for other common DB Web Controls.

Users should already be familiar with creating an ASP.NET project using BDP.NET.

Building the simple ASP.NET application with DB Web Controls and BDP.NET consists of three major steps:

1. Prepare an ASP.NET project with BDP.NET or other connection components.
2. Drag and drop a DBWebDataSource onto the Designer and set its DataSource property to a DataSet, DataView or DataTable.
3. Drag and drop a DBWebGrid and other control onto the Designer.

To prepare an ASP.NET project for DB Web Controls

1. Create an ASP.NET project.
2. Set up BDP.NET or other data access components, setting the DataSource property to an existing DataSet, DataView, or DataTable.

   Tip: For more information about setting up BDP.NET data access components, see the related procedure for building an ASP.NET database application. Instead of using a DataGrid and adding a `DataBind` call, in the following procedure you use DB Web Controls without a `DataBind` call.

To configure a DBWebDataSource

1. Place a DBWebDataSource component on the **Designer**.
2. In the **Object Inspector**, select the DataSource property.
3. Select an existing data source (by default, this is called dataSet1).

To configure DB Web Controls

1. Place a DBWebNavigator component on the **Designer**.
2. In the **Object Inspector**, select a data source in the DBDataSource property drop-down.
3. In the **Object Inspector**, select a DataTable from the TableName property drop-down.

   Tip: If no TableName is available, verify that the BdpDataAdapterActive property is set to **True**.

4. Place a DBWebGrid on the **Designer**.
5. In the **Object Inspector**, select the data source from the DBDataSource property drop-down.
6. In the **Object Inspector**, select a DataTable from the TableName property drop-down.

   The grid displays data.
7. Place other DB Web Controls as needed.
8. Set the values for `DBDataSource`, `TableName`, and other properties as appropriate.

Note: For data-aware Column Controls (such as DBWebTextBox, DBWebImage, DBWebMemo, and DBWebCalendar) additionally set the `ColumnName` property. For data-aware lookup controls (such as DBWebDropDownList, DBWebListBox, and DBWebRadioButtonList), also set the `LookupTableName`, the `DataTextField`, and the `DataValueField` properties.

9 Choose **Run** ▸ **Run**.

The application compiles and the HTTP server displays a Web Form with a DBWebGrid displaying data.

Tip: Dragging web components from the **Tool Palette** places them in an absolute position on an ASP.NET web form. Double-clicking components in the **Tool Palette** leaves them in ASP.NET flow layout. Flow layout is much easier to manage. For instance, controls in an absolute position on a web form can overwrite other controls if they change sizes at runtime. Overwriting might occur when you add rows to and remove rows from a grid control, making the grid control change size.

Building an ASP.NET "Hello World" Application

Though simple, the ASP.NET "Hello World" application demonstrates the essential steps for creating an ASP.NET application. The application uses a Web Form, controls, and an event that will display a result in response to a user action.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.

   The **New ASP.NET Application** dialog box appears.
2. In the **Name** field, enter HelloWorld for the application name.
3. In the **Location** field, accept the default or enter [Inetpub]\HelloWorld, where [Inetpub] is the directory location for IIS projects (for example, C:\Inetpub\wwwroot\HelloWorld).

To change Web server settings (optional)

1. In the **New ASP.NET Application** dialog box, click **View Server Options**.

   The dialog expands to show additional server options.
2. Set the various read and write attributes of the project as needed or accept the defaults.

   Tip: For most ASP.NET projects, the default settings will suffice.
3. Click **OK**.

   The Web Forms Designer appears.

To create the ASP.NET page

1. If necessary, click **Design** view.
2. From the **Web Controls** category of the **Tool Palette**, drag a Button component onto the Designer surface.

   The Button control appears on the Designer. Make sure the control is selected.
3. In **Object Inspector**, set the **Text** property to Hello, world!.

To associate code with the button control

1. In the Designer, double-click the Button control.

   The code-behind Designer appears, cursor in place between event handler brackets.
2. Code the application logic:

```
[C#]
button1.Text = button1.Text + "Hello, developer!";
```

```
[Delphi]
button1.Text := button1.Text + 'Hello, developer!';
```

3 Choose **File** ▸ **Save** to save the application.

To run the "Hello World" application

1 Choose **Run** ▸ **Run**.

The application compiles and the HTTP server displays a Web Form in your default browser with the "Hello, world!" button.

2 Click the "Hello, world!" button.

The server updates the page with the response, "Hello, developer!".

3 Close the Web browser to return to the IDE.

Building an ASP.NET Application

The following procedures describe the general steps required to build a simple ASP.NET project. For more advanced topics, refer to the related information following the procedure.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
   The **New ASP.NET Application** dialog box appears.
2. In the **Name** field, enter the name of your project.
3. In the **Location** field, accept the default path or enter another project path.

   Tip: Most ASP.NET projects reside in the IIS directory Inetpub\wwwroot.

To change Web server settings (optional)

1. In the **New ASP.NET Application** dialog box, click **View Server Options**
   The dialog expands to show additional server options.
2. Set the various read and write attributes of the project as needed or accept the defaults.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.
   The Web Forms Designer appears.

To create an ASP.NET page

1. Make sure the Designer is displayed.
2. From the **Tool Palette**, drag components onto the Designer to define the user interface.
3. Add code-behind logic to components.

To add code-behind logic to a component

1. In the Designer, double-click the component to which you wish to apply logic.
   The code-behind Designer appears, cursor in place between event handler brackets.
2. Add your logic.
3. Run the application.
   The application saves and compiles. Once you compile the application, the generated .aspx file displays HTML in the default web browser.

Building an ASP.NET Database Application

The following procedure describes the minimum number of steps required to build a simple ASP.NET database application using BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of ASP.NET, ADO.NET, and BDP.NET at work.

Building an ASP.NET application with BDP.NET components consists of four major steps:

1. Create an ASP.NET project.
2. Configure BDP.NET connection components and a data source.
3. Add a DataBind call.
4. Connect a DataGrid to the connection components.

Tip: For testing purposes, use the employee.gdb database included with Interbase, if included with your version of the product.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
   The **New ASP.NET Application** dialog appears.
2. In the **Name** field, enter the name of your project.
3. In the **Location** field, enter the project path.

   Tip: Most ASP.NET projects reside in the IIS directory: Inetpub\wwwroot.

To change Web server settings (optional)

1. In the **New ASP.NET Application** dialog, click **View Server Options**
   The dialog expands to show additional server options.
2. Set the various read and write attributes of the project as needed or accept the defaults.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.
   The Web Forms Designer appears.

To configure data components

1. Drag and drop a BdpDataAdapter component onto the Designer. If necessary, select BdpDataAdapter.
2. In **Object Inspector**, select **Configure Data Adapter**.
   The **Data Adapter Configuration** dialog appears.
3. If necessary, select the **Command** tab. From the **Connection** drop-down, select **New Connection**.

4 The **Borland Data Provider: Connections Editor** dialog appears.

Tip: Alternatively, use Data Explorer to drag and drop a table on to the Designer surface. Data Explorer sets the connection string automatically.

To set up a connection

1 In **Borland Data Provider: Connections Editor**, select the appropriate item from the **Connections** list.

2 In **Connection Settings**, enter the **Database** path.

Note: If referring to a database on the local disk, prepend the path with localhost:. If using Interbase, for example, you would enter the path to your Interbase database: localhost:C:\Program Files \Borland\Interbase\Examples\Database\employee.gdb (or whatever the actual path might be for your system).

3 Complete the **UserName** and **Password** fields for the database as needed.

4 Click **Test** to confirm the connection.

A dialog appears confirming the status of the connection.

5 Click **OK** to return to the **Borland Data Provider: Connections Editor** dialog.

6 Click **OK** to return to the **Data Adapter Configuration** dialog.

In the **Command** tab, the areas for **Tables** and **Columns** are updated with information from your connection.

To set a command

1 In the **Select** area, enter an SQL command.

Tip: For Interbase's employee.gdb database, you might enter select * from SALES, as an example.

2 Click the **Preview Data** tab.

3 Click **Refresh**.

Column and row data appear.

4 Click the **DataSet** tab.

5 Confirm that **New DataSet** is selected.

6 Click **OK**.

New components for DataSet and BdpConnection appear on the Designer.

7 Select BdpDataAdapter component.

8 In **Object Inspector**, select the Active property drop-down and set the value to **True**.

To connect a DataGrid to a DataSet

1 Drag and drop a DataGrid web control onto the Designer. If necessary, select DataGrid.

2 In **Object Inspector**, select the DataSource property drop-down. Select the DataSet component that you generated previously (the default is DataSet1).

3 In **Object Inspector**, select the DataMember property drop-down. Select the appropriate table.

The DataGrid displays data from the DataSet.

To add a DataBind call

1. Use the **Object Inspector** drop-down to select the Web Form (**WebForm1** is the default).
2. In **Object Inspector**, select the **Events** tab.
3. Set the Load event to **Page_Load**.
4. In **Object Inspector**, double-click **Page_Load**.

   The code-behind Designer appears, cursor in place between event handler brackets.
5. Code the DataBind call:

```
[C#]
this.dataGrid1.DataBind();
```

```
[Delphi]
Self.dataGrid1.DataBind();
```

   Note: If you are using data aware controls, for instance from a third-party provider, you may not need to code the DataBind call.
6. Choose **Run** ▸ **Run**.

   The application compiles and the HTTP server displays a Web Form with the datagrid.

While presenting a minimum number of steps required to build a database project, the preceding procedure demonstrates the major components of the ASP.NET, ADO.NET, and BDP.NET architectures at work, including: providers, datasets, and adapters. The adapter connects to the physical data source via a provider, sending a command that will read data from the data source and populate a dataset. Once populated, a datagrid displays data from the dataset.

Once created, use other BDP.NET designers to modify and maintain the components of your project.

Converting HTML Elements to Server Controls

Unlike Web controls, HTML elements can not, by default, be controlled programmatically. However, you can convert an HTML element to a server control and then write code to access or modify the element. Most of the HTML elements that appear in the **Tool Palette** can be converted by using the **Run As Server Control** command. HTML elements that do not appear on the **Tool Palette**, such as `body`, can be converted manually.

The following procedures explain how to convert an HTML `table` element by using the **Run As Server Control** command, and how to convert a `body` element manually.

To convert an HTML table element to a server control

1. With an ASP.NET application open, display the Designer.
2. From the **Tool Palette**, add the **HTML Table** element from the **HTML Elements** category to the Designer.
3. Right-click the **Table** element on the Designer and choose **Run As Server Control**.

   The server control icon is added to the **Table** element. In the .aspx file, the `id="TABLE1"` and `runat="server"` attributes are added to the `table` tag. In the code-behind file, `TABLE1` is declared using System.Web.UI.HtmlControls.HtmlTable.
4. You can now reference `TABLE1` in your code. To demonstrate this, add a **Button** from the **Web Controls** category of the **Tool Palette** to the Designer.
5. Double-click the button. The **Code Editor** opens and is positioned at the click event for the button.
6. Add the following code to the event handler to change the background color of the table to blue. Note that `TABLE1` is the id that was added automatically to the the `table` tag in Step 3.

```
[Delphi]
TABLE1.BgColor := 'blue';
```

```
[C#]
TABLE1.BgColor = "blue";
```

7. Choose **Run** ▸ **Run** to run the application.
8. Click the button to change the table color.

To convert an HTML body element to a server control manually

1. With an ASP.NET application open, display the .aspx file.
2. Add the `runat="server"` and `id="identifier"` attributes to the `body` tag, where identifier is a descriptive identifier, such as `bodytag`.
3. Add the following declaration to the strict protected section of the code-behind file:

```
[Delphi]
bodytag: System.Web.UI.HtmlControls.HtmlGenericControl;
```

```
[C#]
protected System.Web.UI.HtmlControls.HtmlGenericControl bodytag;
```

4. You can now reference `bodytag` in your code. To demonstrate this, add a **Button** from the **Web Controls** category of the **Tool Palette** to the Designer.

5 Double-click the button. The **Code Editor** opens and is positioned at the click event for the button.

6 Add the following code to change the background color of the Web Form to yellow.

```
[Delphi]
bodytag.Attributes['bgcolor'] := 'yellow';
```

```
[C#]
bodytag.Attributes["bgcolor"] = "yellow";
```

7 Choose **Run** ▶ **Run** to run the application.

8 Click the button to change the background color of the form.

Creating a Briefcase Application with DB Web Controls

You can use DB Web Controls, XML caching, and the BDP.NET data adapters to create server-side briefcase applications. You can only create this type of application when using user authentication, to guarantee that each user has a unique copy of the XML file.

To create a briefcase application

1. Create a BDP.NET application.
2. Add a DBWebDataSource control and link to the BDP DataSet.
3. Configure the DBWebDataSource control to generate XML and XSD files.
4. Configure the AutoUpdateCache and UseUniqueFileName properties.
5. Configure an OnApplyChangesRequest to call the BdpDataAdapterAutoUpdate method.
6. Run the application.

To configure the AutoUpdateCache and UseUniqueFileName properties

1. Build a standard ASP.NET database application using the BDP.NET components and the DBWebDataSource component.
2. Specify XML and XSD filenames for non-existent files in the DBWebDataSource component.

   Note: It is best to create these files in the project directory or in a subdirectory of the project directory, typically on your web server.

3. Set AutoUpdateCache to **True**.
4. Set UseUniqueFileName to **True**.
5. Select the **Events** tab for the DBWebDataSource component.
6. Double-click the OnApplyChangesRequest field to display the event handler in the **Code Editor**.
7. Add the following code:

```
BdpDataAdapter1.AutoUpdate;
```

8. Choose **Run** ▸ **Run**.

   The first time the application runs, it creates the XSD file using the server metadata.

The first time a user runs the application, the application retrieves data from the server. When the user changes data, thereafter, the application saves those changes to the server in a unique filename based on the username. If the user shuts down the application and runs it again at a later time, the application restores the user's specific data. At this point, the user can undo or modify the data. Anytime the OnApplyChangesRequest is called successfully, the application deletes the unique user files and creates new ones.

Warning: If the tables or columns accessed by the application are altered after the application has run, you must delete the XSD file to avoid a mismatch between the XSD file and the server metadata. Otherwise, you can experience runtime errors and unpredictable behavior.

Creating a Virtual Directory

When you create an ASP.NET application, the IDE automatically creates a virtual directory for you based on the settings in the **New ASP.NET Application** dialog box.

However, the IDE can also create a virtual directory for an application that you did not create within the IDE, such as the demo applications found in the DBWeb folder (located by default at C:\Program Files\Borland\BDS\4.0\Demos\Delphi.Net).

To create a virtual directory for an existing application

1. Open the ASP.NET application project file in the IDE.
2. Choose **Project** ▸ **Options** ▸ **Debugger** ▸ **ASP.NET**.

   The default application settings are displayed. Accept the default settings or change them as needed.
3. If you are creating a virtual directory for use with Internet Information Server (IIS), click the **Server Options** button to display the **Configure Virtual Directory** dialog.

   If you change the name of the virtual directory or its alias, you can also change the permissions associated with the virtual directory.
4. Click **OK** to return to the project options.
5. Click **OK** to exit the project options.

The virtual directory is created for you, enabling you to run the application.

Creating an XML File for DB Web Controls

You can use XML files as your data source, particularly if you want to prototype applications without reading from and writing to a database. First you must create the XML file. The DBWebDataSource control provides a powerful way to create the XML file based on real database data. This procedure assumes that you can create a connection to a live database containing the data you want to use.

To create and use an XML file

1. Create an ASP.NET application using DB Web Controls.
2. Specify the XML file as a data source for a new ASP.NET application.

To create an ASP.NET application using DBWeb Controls

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Create a database connection and data adapter using the BDP.NET controls or other data adapter controls.
3. Drag and drop a DBWebDataSource control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
4. In the XMLFileName property or in the XMLSchemaFile property, specify a new file name of a file that does not yet exist.
5. Generate a DataSet from the data adapter.
6. Set the DataSource property of the DBWebDataSource to dataSet1.
7. Set the Active property of the data adapter to **True**.
8. Choose **Run** ▸ **Run**.

   This runs the application but also creates the XML file or XSD file and fills it with data from the DataSet.

To specify the XML file as a data source for a new ASP.NET application

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Drag and drop a **DataSet** component onto the **Designer** from the **Data Components** area of the **Tool Palette**.
3. Drag and drop a DBWebDataSource control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
4. Specify the existing XML file name in the XMLFileName property of the DBWebDataSource control.

   Note: If you created an XSD file instead of an XML file, you specify the XSD file name in this step.
5. Specify the DataSet component in the DataSource property of the DBWebDataSource control.
6. Drag and drop a DBWebGrid control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
7. Set the DBDataSource property of the DBWebGrid to the name of the DBWebDataSource
8. Choose **Run** ▸ **Run** to display the application.

   The application pulls data from the DataSet and XML file to fill the DBWebGrid.

Warning: It is possible for you to specify an existing XML file in the XMLFileName property of your DBWebDataSource along with an active BdpDataAdapter and its DataSet. You can run the application and the DBWeb controls will display the data from the XML file. However, this is not the intended use or behavior of the XML capabilities of the DBWebDataSource. Although your XML file data may display properly, the results of an update or any other operations on the data will be unpredictable.

Creating Metadata for a DataSet

When you choose to use an XML file for a data source in an ASP.NET application using DB Web Controls, you may need to create the metadata to structure the XML data in your DataSet. If you chose to create an XML file without an XML schema file (.xsd), you need to manually create the metadata. This procedure assumes that you have already created an XML file containing data.

To set up the application

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Drag and drop a DBWebDataSource control onto the form.
3. Drag and drop a DataSet component onto the form.
4. Click the ellipsis button (...) next to the XMLFileName property of the DBWebDataSource and locate your XML file.
5. Select the DataSet component in the **Component Tray**.
6. Click the **Tables (Collection)** property to display the **Tables Collection Editor**.

To create the metadata

1. Click **Add** to add a new table to the collection.

   For the sake of illustration, we'll use the following XML records.

```
<?xml version="1.0" standalone="yes"> /// XML Declaration
<NewSongs>

 /// <song> becomes the table name in your DataSet.
    <song>

    /// <songid> becomes Column1 in your DataSet.
        <songid>1001</songid>

 /// <title> becomes Column2 in your DataSet.
        <title>Mary Had a Little Lamb</title>
    </song>
    <song>
        <songid>1003</songid>
        <title>Twinkle, Twinkle Little Star</title>
    </song>
</NewSongs>
```

2. Change the **TableName** property to song.
3. Click the **Columns (Collection)** property to display the **Columns Collection Editor**.
4. Click **Add** to add a new column.
5. Change the **ColumnName** property to songid.
6. Click **Add** to add another new column.
7. Change the **ColumnName** property to title.
8. Click **Close** to close the **Columns Collection Editor**.
9. Click **Close** to close the **Tables Collection Editor**.

   You have now created the metadata to match the XML file data.

Debugging and Updating ASP.NET Applications

During the installation of Developer Studio 2006, the install program requested permission to update the machine.config file on your computer. This information is necessary for debugging Developer Studio 2006 applications under IIS. If you replied **Yes** to that prompt, Borland debugger information was written to machine.config and will be available to the applications that you created with Delphi 8. **You need not perform this procedure.**

If you replied **No** to that prompt, the debugger information is written to the application web.config file when you create an ASP.NET application with Developer Studio 2006. However, you will need to add this information manually to web.config for applications that were created with Delphi 8. Otherwise, attempting to debug your Delphi 8 application with Developer Studio 2006 may result in the following error:

Unable to start debugging on the web server. Unable to attach to ASP.NET worker process (typically aspnet_wp.exe or w3wp.exe).

To update the web.config file for a Delphi 8 ASP.NET application

1. Open the web.config file in the IDE or a text editor.
2. Replace the following lines:

```
<compilation
     debug="true"
     defaultLanguage="c#">
</compilation>
```

   with this:

```
<compilation defaultLanguage="c#" debug="true">
    <assemblies>
      <add assembly="Borland.dbkasp, Version=9.0.0.1,
                     Culture=neutral, PublicKeyToken=b0524c541232aae7"/>
    </assemblies>
</compilation>

<httpModules>
   <add name="DbgConnect" type = "Borland.DbkAsp.DbkConnModule,
                           Borland.dbkasp,Version=9.0.0.1, Culture=neutral,
                           PublicKeyToken=b0524c541232aae7"/>
</httpModules>
```

3. Save the web.config file.
4. Open the application project in the IDE and run it.

Note: Before deploying an ASP.NET application, you should disable debugging and remove debugger references from the web.config file, as described in the topic listed below.

Generating HTTP Messages in ASP.NET

When attempting to debug your ASP.NET applications, you may find that the error messages are cryptic or even meaningless. This may be the result of having a specific option set in your Internet Explorer browser. To assist your debugging efforts, you should change this option.

To generate more meaningful error messages

1. In Internet Explorer (assuming you are using IE) choose **Tools** ▸ **Internet Options**.
2. Click the **Advanced** tab.
3. Deselect the **Show friendly HTTP error messages** check box.
4. Click **OK**.

   This turns off friendly messages and provides meaningful ASP.NET messages.

Modifying Database Connections

The basic elements of a connection string tend to be the same from one database type to another. However, each database type supports slightly different connection string syntax. This topic addresses those differences.

To modify different types of database connections

1. Click on the **Data Explorer** tab in the IDE.
2. Select the database type of your choice.
3. Right-click to display the popup menu.
4. Choose **Modify Connection** to display the **Connections Editor**.

   The properties in the Connections Editor are organized into three categories: Connections, Options, and Provider Settings. The Connections options designate the database and authentication parameters. The Options area includes various database-specific database options, including transaction isolation types. The Provider Settings area specifies assemblies and the client libraries required to accomplish the connection to the given database.

> **Note:** All of the procedures in this topic assume that you already have installed a database client, server, or both, and that the database instance is running.

To modify an InterBase connection

1. Either enter the database name or navigate to the database on your local disk or a network drive, by clicking the ellipsis button to browse.

   The standard supplied databases are typically installed into `C:\Program Files\Common Files\Borland Shared\Data`.
2. Enter the password and username.

   By default, these are `masterkey` and `sysdba`, respectively.
3. Set the following options, if necessary.

   The default values are shown in the following table.

Option	Description	Default
CommitRetain	Commits the active transaction and retains the transaction context after a commit.	False
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
RoleName	If there is a role for you in the database, you can enter the rolename here. The role is generally an authentication alias, that combines your identify with your access rights.	myRole
ServerCharSet	Specifies the character set on the server.	—
SQLDialect	Sets or returns the SQL dialect used by the client.	3
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting	ReadCommitted

	in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	
WaitOnLocks	Specifies that a transaction wait for access if it encounters a lock conflict with another transaction.	False

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Interbase,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Interbase
VendorClient	gds32.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

> **Note:** If you are writing ASP.NET applications, and are running the ASP.NET Web Forms locally for testing purposes, you might need to modify the path statement that points to your database, to include the `localhost:` designation. For example, you would modify the path shown earlier in this topic as such: `localhost:C:\Program Files\Common Files\Borland Shared\Data\employee.gdb`.

Note: Your connection string should resemble something like

```
database=C:\Program Files\Common Files\Borland Shared\Data\EMPLOYEE.GDB;
assembly=Borland.Data.Interbase,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=gds32.dll;provider=Interbase;username=sysdba;password=masterkey
```

To modify an MS SQL Server connection

1 Enter the database name in the **Database** field of the **Connections Editor**.

For example, use one of the sample MS SQL Server databases, such as Pubs or Northwind. There is no need to add the file extension to the name.

2 Enter the hostname.

If you are using a local database server, enter `(local)` in this field.

3 If you are deferring to your OS authentication, set **OSAuthentication** to **True**.

4 If you are using database authentication, enter the password and username into the appropriate fields.

By default, the SQL Server database username is `sa`.

5 Change the database options if necessary.

The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False

QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

6 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Mssql,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	MSSQL
VendorClient	sqloledb.dll

7 Click **Test** to see if the connection works.

8 Click **OK** to save the connection string.

Note: If you are writing ASP.NET applications, and are running the ASP.NET Web Forms locally for testing purposes, you might need to modify the path statement that points to your database, to include the `localhost:` designation, prepended to the path.

Note: Your connection string should resemble something like

```
assembly=Borland.Data.Mssql,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=sqloledb.dll;osauthentication=True;database=Pubs;username=;hostname=(local);
password=;
provider=MSSQL
```

To modify a DB2 connection

1 Enter the path to the database.

2 Enter the password and username into the appropriate fields.

3 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Db2,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	DB2
VendorClient	db2cli.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

To modify an Oracle connection

1 Enter the path to the database.

2 If you are deferring to your OS authentication, set **OSAuthentication** to **True**.

This means that the system defers to your local system username and password to login to the database.

3 If you are using database authentication, enter the password and username into the appropriate fields.

For example, the typical Oracle username and password for the sample database is `SCOTT` and `TIGER`, respectively.

4 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

5 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Oracle,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Oracle
VendorClient	oci.dll

6 Click **Test** to see if the connection works.

7 Click **OK** to save the connection string.

To modify an MS Access connection

1 Either enter the database name or navigate to the database on your local disk or a network drive, by clicking the ellipsis button to browse.

If you have the Office Component Toolkit installed, you might find Northwind in `C:\Program Files\Office Component Toolpack\Data\Northwind.mdb`.

2 Enter the username and password.

By default, you can generally try `admin` for the username and leave the password field empty.

3 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Msacc,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	MSAccess
VendorClient	msjet40.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

Note: Your connection string should resemble something like

```
database=C:\Program Files\Office Component Toolpack\Data\Northwind.mdb;
assembly=Borland.Data.Msacc,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=msjet40.dll;provider=MSAccess;username=admin;password=
```

To modify a Sybase connection

1 Enter the path to the database.

2 Enter the password and username into the appropriate fields.

3 Set the following database options, if necessary. The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
ClientAppName	Client application name set by the middle-tier application.	—

ClientHostName	Client host name set by the middle-tier application.	—
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
PacketSize	Specifies the number of bytes per network packet transferred from the database server to the client.	512
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Sybase,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Sybase
VendorClient	libct.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

Note: Your connection string should resemble something like

```
assembly=Borland.Data.Sybase,Version=2.0.0.0,Culture=neutral,
PublicKeyToken=91d62ebb5b0d1b1b;vendorclient=libct.dll;database=Pubs;
username=admin;hostname=host1;password=;provider=Sybase
```

Porting a Delphi for Win32 Web Service Client Application to Delphi for .NET

The following steps are required to port your Delphi for Win32 Web Services client application to Delphi for .NET.

To port your web service

1. Change the existing RIO form components.
2. Change the uses clause.
3. Add a web reference.
4. Change the web service invocation code.

To change your existing form components

1. Copy and save the web reference URL from your existing RIO component.
2. Delete the HTTPRio component from the form if it was not dynamically created.

To change the uses clause

1. Remove any Delphi for Win32 SOAP units from the clause.

   These include, but are not restricted to InvokeRegistry, RIO, and SOAPHTTPClient.

   Warning: The preceding list of units is not inclusive. Make sure you identify all SOAP units, regardless of naming convention. Not all of the units include the word SOAP in the name.

2. Remove the reference to the Delphi for Win32 WSDL Importer-generated Interface proxy unit.
3. Remove the proxy unit from the project.

To add a web reference

1. Open a Delphi for Win32 project in Developer Studio 2006 and choose **Project** ▸ **Add Web Reference**.

   Once you have saved the project, the UDDI Browser appears.

2. Enter the URL you want to use, either a service you are already familiar with, or the one saved from your RIO component into the list box at the top of the Browser.

   Note: If you want to locate a WSDL file on your local disk, you can click the ellipsis button next to the list box and search for the document. You can also navigate to one of the web service sites listed in the UDDI Browser if you want to use a published service.

3. Click the **Add Reference** button to add the WSDL document to your project.

   Developer Studio 2006 creates the necessary web reference and the corresponding proxy unit based on the WSDL document. A new Web References node appears in the **Project Manager**. Expand it to see the associated WSDL and proxy code files.

4. Choose **File** ▸ **Use Unit**.

To change the web service invocation code

1 In the code file for your application, locate the code that invokes the web service.

Assume it looks something like this:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
    HelloService: Service3Soap;
begin

    // The next line will be slightly different if you have
    // used a component or generated the method dynamically.

 // This is how it will look if you used a component.
    HelloService := (HTTPRIO1 as Service3Soap);

    // This is how it will look if created dynamically.
    // GetService3Soap is the global method in the proxy unit.
 HelloService := GetService3Soap;

    Caption := HelloService.HelloWorld;
end;
```

2 Change the var section from this:

```
[Delphi]
var
// This is the type of the old proxy interface.
    HelloService: Service3Soap;
```

to

```
[Delphi]
var
// This is the type of the new proxy class.
    HelloService: Service3;
```

This assumes the name of your service is Service3. Change the name accordingly.

> **Note:** You will see that what was formerly created as an interface is now created as a class. The .NET Framework provides automatic garbage collection, and so certain restrictions placed on the use of classes in previous versions of Delphi may no longer apply when using Developer Studio 2006.

3 Change the first line in the procedure block from this:

```
[Delphi]
HelloService := (HTTPRIO1 as Service3Soap);
```

to:

```
[Delphi]
HelloService := Service3.Create;
```

The updated code should look like this:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
    HelloService: Service3;
begin
    HelloService := Service3.Create;
    Caption := HelloService.HelloWorld;
end;
```

Your code is most likely more complex than this example. However, these instructions cover the basic steps for porting any Delphi for Win32 application that uses web services to Developer Studio 2006.

Setting Permissions for XML File Use

You need to grant rights to clients who will be using your ASP.NET applications, if you want to avoid a permissions error when using an XML file as a data source. There are two ways to do this, as described in the following procedures.

To give users rights when the UseUniqueFileName property is false

1. Right-click the Windows Start menu and choose **Explore**.
2. Choose **Tools** ▸ **Folder Options**.
3. Choose the **View** tab.
4. Uncheck the **Use Simple File Sharings** option.
5. Click **Apply to All Folders**.
6. Click **OK**.
7. Locate the XML file being used in the project, then right-click and select **Properties**.
8. If available, select the **Security** tab.
9. Add user **EveryOne** and set **Full Rights** to the file.

To give users rights when UseUniqueFileName is true and user authentication is in use

1. On the Windows Control Panel **User Accounts** dialog, create a new user.
2. In the IIS virtual directory where your web application is built, create a new folder named CacheFiles.
   Typically, your IIS virtual directories are in the C:\Inetpub\wwwroot directory.
3. Using the Windows Explorer, located the folder CacheFiles.
4. Right-click and choose **Properties**.
5. Choose the **Security** tab and add the user you created in Step 1.
6. Add **Full Rights** to the folder.
7. Move the XML file to this folder.
8. Set the XMLFileName property of the DBWebDataSource in your application to this file.

Note: You must make sure that the **Use Simple File Sharings** option in your Windows **Folder Options** is unchecked.

Setting Up a Cassini Web Server

Borland distributes the CassiniWebServer as a demo, so that Delphi users can utilize it as an alternative to the Microsoft IIS web server during development. This is because it is easy to set up, and it works well with the Delphi debugger. Microsoft makes this managed code web server available as a free download. The following procedures describe the general steps required to download and configure a Cassini Web Server.

To configure the CassiniWebServer:

1 **Tip:** .

To xxxxxxx

To xxxxxx

To xxxxxx

Troubleshooting ASP.NET Applications

Unlike traditional window-based applications, web applications are dependent on servers and resources that are not directly within the control of the application or the user. Web applications are often hybrid combinations of client, server, and network resources.

The areas you need to check include ASP.NET installation, IIS installation and configuration, and security. All three of these areas are extensive and complex. The following procedures provide solutions to some of the most common problems.

Note: The following suggestions apply only to IIS 5.1.

To troubleshoot your ASP.NET application

1. Install or reinstall ASP.NET.
2. Create or check your ASP.NET user account.
3. Install or reinstall IIS.
4. Start or restart IIS.
5. Configure IIS to recognize your application.
6. Add document types to IIS.
7. Set anonymous authentication.
8. Check your database connection, if applicable.

To install or reinstall ASP.NET

1. Choose **Start** ▸ **Run** to display the **Run** dialog box.
2. Type cmd /e in the **Open** drop down list box.
3. Click **OK**.
4. Change directories to c:\Windows\Microsoft.NET\Framework\v1.1.4322.
5. Enter the command aspnet_regiis.exe -i.
6. Press Enter.

> **Note:** If you want to know the various command flags for the aspnet_regiis.exe utility, follow the basic command with a ? character instead of the -i flag.

To create or check your ASP.NET user account

1. Choose **Start** ▸ **Control Panel** ▸ **User Accounts** to display the list of user accounts on your system.
2. If you do not have an ASPNET user account, create one.
3. Restart your machine.

> **Warning:** Do not give your ASPNET user administrator privileges. This opens up a security hole in your system and makes deployed ASP.NET applications vulnerable to hacking. Instead, create an impersonated user.

To install or reinstall IIS

1. Choose **Start** ▸ **Control Panel** ▸ **Add or Remove Programs**.
   This displays the **Add or Remove Programs** dialog box.
2. Click **Add/Remove Windows Components**.
   This displays the **Windows Components Wizard**.
3. Check the **Internet Information Services (IIS)** check box.
4. Click **Next**.
5. Click **Finish**.
6. Start IIS.

To restart IIS

1. Choose **Start** ▸ **Control Panel** ▸ **Administrative Tools** ▸ **Internet Information Services**.
2. Select the local computer node.
3. Right-click and select **Restart IIS...**.
   This displays the **Stop/Start/Reboot** dialog.
4. Choose the task you want to accomplish from the drop down list box.
5. Click **OK**.

To configure IIS to recognize your application

1. In the IIS console, locate the folder or virtual directory containing your web application.
   If there is not a folder or virtual directory, you will need to create a virtual directory.
2. Select the folder.
3. Right-click and select **Properties**.
4. Click the **Virtual Directory** tab.
5. Under the **Application Settings** area, click the **Create** button.
   If the **Remove** button is displayed instead, you can remove, then create the virtual directory again, if necessary.

To add document types to IIS

1. Choose **Start** ▸ **Control Panel** ▸ **Administrative Tools** ▸ **Internet Information Services**.
2. Select **Default Web Site**.
3. Right-click and select **Properties**.
4. Click the **Documents** tab.
5. Click **Add**.
   This displays the **Add Default Document** dialog box.
6. Add WebForm1.aspx in the **Default Document Name** textbox.
7. Click **OK** twice.

To set anonymous authentication

1. In the IIS console, locate the folder or virtual directory containing your web application.
   If there is not a folder or virtual directory, you will need to create a virtual directory.
2. Select the folder.
3. Right-click and select **Properties**.
4. Click the **Directory Security** tab.
5. Click **Edit**.
6. Select the **Anonymous Access** check box.
7. In the **User name:** field, enter the name of the ASPNET user you created.
8. Check the **Integrated Windows authentication** check box or add your own password.
9. Click **OK** twice.

To check your database connection

1. Click the **Data Explorer** tab to display your database connections.
2. Expand the provider list to display a valid database connection.
3. Right-click and choose **Modify Connection**.
   This displays the **Connections Editor**.
4. If the Database connection string does not contain the localhost specifier, prepend it to the connection string, as in the following example:

```
localhost:C:\Program Files\Common Files\Borland Shared\Data\EMPLOYEE.GDB
```

5. Make sure all of your other connection options are set property.
6. Click **Test** to make sure the connection is alive.

Using the ASP.NET Deployment Manager

You can add an ASP.NET Deployment Manager to an ASP.NET application project to assist you with deploying the application. The Deployment Manager determines which files are required for deployment, requests the destination directory name and connection information, and then copies the files to the destination directory. The Deployment Manager generates a list of files to copy based on the names of the files in your project directory, but you can include or exclude files as needed.

You can use the right mouse button, when the Deployment Manager window is displayed, to see options for displaying, copying, deleting, modifying, and filtering destination files.

When the **Show Assembly References** option is enabled, the Deployment Manager window displays all of the assemblies referenced by the project. The system assemblies are shown, but disabled (grayed). These disabled assemblies can't be deployed.

The **External Files....** option allows you to pick the external files that you want to deploy. A dialog box with a check list box is pre-populated with the BDP database libraries, since one of these often needs to be deployed. You can also add files to the list using a **File** ▸ **Open** dialog. The list box has a column that indicates the destination subdirectory for the external file. You can edit the destination path. The files that are checked when you click OK will be shown in the Deployment Manager.

See the links at the end of this topic for more information about the right-click options for the Deployment Manager.

Considerations

- To enable IIS debugging of Developer Studio 2006 applications , during the installation of Developer Studio 2006, the install program requested permission to update the machine.config file on your computer. If you rep lied **Yes** to that prompt, Borland debugger information was written to machine.config. If you replied **No** to that prompt, that debugger information is written to the application web.config file when you create an ASP.NET application with Developer Studio 2006. Before deploying the application, you should disable debugging to optimize the application, as described in the following procedure. Additionally, if you chose not to update machine.config, you should remove references to the Borland debugger modules in web.config, because those modules might not be available on the deploy target computer.
- Consider maintaining a separate web.config file for deployment purposes. For example, you might maintain a file named web.config.deploy and rename it to web.config during deployment. Use the Deployment Manager **Change Destination Filename** command to rename the file.
- You can create the destination directory while using the Deployment Manager, however, you will then need to use IIS to create the virtual directory before using the application. Alternatively, you can deploy to an existing virtual directory.
- When deploying to an FTP site, the Deployment Manager will retain your FTP connection information. You may save your FTP connection password, however, it will be saved as unencrypted, plain text.
- You can add multiple Deployment Managers to an ASP.NET project and configure them to deploy to different destination directories.
- Some of the commands that are available in the Deployment Manager are also available in the **Project Manager** context menu.

To remove debugger references in the web.config file

1. In the IDE or a text editor, open the web.config file that you will use for the deployed ASP.NET application.
2. In the `<compilation>` section, change `debug="true"` to `debug="false"`.
3. Skip this step if you chose to update machine.config during the installation of Developer Studio 2006 (see the **Considerations** above for details).

   Remove or comment out the following references to the Borland debugger assembly and modules:

```
<assemblies>
     <add assembly="Borland.dbkasp, Version=9.0.0.1,
                           Culture=neutral, PublicKeyToken=b0524c541232aae7"/>
</assemblies>

<httpModules>
     <add name="DbgConnect" type =
                           "Borland.DbkAsp.DbkConnModule,Borland.dbkasp,Version=9.0.0.1,
Culture=neutral,
                           PublicKeyToken=b0524c541232aae7"/>
</httpModules>
```

4 Save the file and recompile the application.

To deploy an ASP.NET application

1 In the IDE, open the ASP.NET application project to be deployed.

2 Choose **File** ▸ **New** ▸ **Other** ▸ **Deployment** ▸ **ASP.NET Deployment** and click **OK**. (The **Deployment** node is not displayed in the **New Items** dialog box unless an ASP.NET project is open.)

The **Deploy** tab is displayed and a .bdsdeploy file is added to the project directory and displayed in the **Project Manager**. The files required for deployment are listed on the left side of the **Deploy** tab under **Source Files**.

> **Tip:** Only files that have been saved are displayed in the list; save any new files and refresh the Deployment Manager to display the files.

3 In the **Destination** drop-down list, select either **Folder Location** or **FTP Location**.

If you select **Folder Location**, the **Browse For Folder** dialog box is displayed. You can select an existing directory or click **Make New Folder** to create a new one.

If you select **FTP Location**, the **FTP Site** dialog box is displayed. Enter the connection information. Click **Help** for an explanation of each field.

Click **OK** to return to the Deployment Manager.

4 If you selected an FTP location, check the **Connected** check box to connect and display the files, if any, in the destination directory.

5 Review the files in the **Source Files** list.

Click a file to display detailed file information in the text box below the file list.

6 To copy all of the files to the destination directory, click the **Copy All New or Modified Files to Destination** button on the toolbar at the top of the Deployment Manager. The files are copied immediately to the destination directory and displayed in the **Destination Files** list.

To modify the file list, right-click anywhere in the file list and use the context menu commands, or use the file list status buttons, as described below.

> **Tip:** To select a file in the list, click the file name. To select multiple files, press CTRL and click the files. To select a range of files, press CTRL+SHIFT, click the first file in the range and then the last file in the range.

Context Menu Command	Description
Refresh	Redisplays the Deployment Manager to reflect changes in the file lists.

Copy Selected File(s) to Destination	Copies the selected files to the destination directory.
Delete Selected Destination File(s)	Deletes the selected files from the destination directory after displaying a confirmation prompt for each file.
Change Destination Filename	Displays a dialog for renaming the selected file in the destination directory.
Copy All New and Modified Files to Destination	Copies all of the files marked with to the destination directory. This command is also available on the Deployment Manager toolbar and by right-clicking the .bdsdeploy node in the **Project Manager**.
Delete All Destination Files Not in Project	Deletes any of the files marked with from the destination directory after displaying a confirmation prompt for each file.
Show Ignored Groups and Files	Displays all of the files in the project directory, even those that are not required to deploy the application.
Ignore Group(s)	Causes the selected file to be ignored by the Deployment Manager.
Ignore File(s)	Causes all of the files in a node of the source files list to be ignored by the Deployment Manager.
Enable Logging	Logs the operations performed by the Deployment Manager in a file named DeployLog.txt in the project directory.
View Log	Displays the log file in the default text editor.

7 When you are satisfied with the deployment criteria, save your changes to the .bdsdeploy file.

When you reopen the project, you can open the Deployment Manager from the **Project Manager** and deploy the application as is, or modify the deployment criteria as described above.

The following buttons indicate the status of the files in the file list and can be used to copy or delete the file, as described below.

File List Status Button	Description
	The file is elligible to copy (it does not exist in the destination directory, or the source file has changed since it was last copied to the destination). Click the button to copy the file to the destination directory.
	The file exists in the destination directory, but not in the project directory. You can probably safely delete it from the destination directory. Click the button to delete the file from the destination directory.
	The status of the file in the is unknown. It might have a later time stamp than the file in the project directory. Click the button to replace the file in the destination directory.

To create an IIS virtual directory for a new destination directory

1 Open IIS on the computer where you deployed the application.

On Windows XP, for example, choose **Start** ▸ **Control Panel** ▸ **Administrative Tools** ▸ **Internet Information Services**.

2 In the **Internet Information Services** dialog box, expand the tree view to display the local computer node.

3 Right-click the **Default Web Site** node and choose **New** ▸ **Virtual Directory**.

The **Virtual Directory Creation Wizard** is displayed.

4 Follow the prompts on each page of the wizard to create the virtual directory.

For more information about virtual directories, refer to the IIS online Help system.

Using the DB Web Control Wizard

The **DB Web Control Wizard** helps you create a data-aware web control based on a standard web control.

To start the DB Web Control Wizard

1 Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **DB Web Control Library**.

> **Note:** You can also use the separate **DB Web Control Wizard** for C#. It works identically to the wizard described here.

This displays the **New DB Web Control Wizard**.

2 Enter a name for the control in the **Control Name** textbox.

3 Select **Bind to DataTable**.

This informs the wizard to add to the control file code that implements IDBWebDataLink. This interface defines the means to access data source and table information.

4 Select **Bind to DataColumn** if you want to bind to a column, for instance, if your control supports a single type of data.

This informs the wizard to add to the control file code that implements IDBWebColumnLink. This interface defines the means to access a column in the table accessed by way of IDBWebDataLink.

5 If you select **Bind to DataColumn** and your control is one of the lookup controls, such as a listbox, radio button group, or check box control, and you want the new control to be a lookup control also, check the **Supports Lookup** check box.

This informs the wizard to add to the control file code that implements IDBWebLookupColumnLink. This interface defines the means to access the lookup table, the text field and value field of the column accessed by way of IDBWebColumnLink.

The **DB Web Control Wizard** creates a template file and displays it in the **Code Editor**. You then modify this file to inherit from a specific DB Web control.

Using the HTML Tag Editor

When you are creating or editing an HTML file, you can use the **Tag Editor**window, beneath the Designer. The **Tag Editor** lets you review and modify HTML tags while viewing the corresponding controls in the Designer window, above it. The **Tag Editor** allows you to use the **Code Completion**, **Error Insight**, and **Code Template Completion** features that are also available in the **Code Editor**. Refer to the links at the end of this topic for more information about using each of these features.

The**Tag Editor** works with one tag at a time, unless you have the Document object selected or you have zoomed out from a tag. (You'll see the item "DOCUMENT" on the Object Inspector when the document object is selected).

The zoom buttons allow you to zoom out to a tag's parent and zoom back in to the selected child tag. Zooming isn't specific to the tag, it's more generic to the markup in the document itself. For example, if the cursor is on a tag in your HTML markup, and you use the Zoom command, it will take you to the outer tag, or one level above the attribute where the cursor is positioned.

Validation against standard HTML style rules occurs automatically. If validation fails, the incorrect element is highlighted in red in the **Designer** and **Error Insight** will appear in the **Tag Editor** to help you correct the problem.

To view HTML code for an individual control

1. With the Designer displayed, drag an HTML element from the **Tool Palette** to the Designer surface.

   The **Tag Editor** displays the HTML code.

2. To view the individual control's code, click anywhere on the Designer surface to deselect the control.

   The HTML code appears in the tag editor window, with syntax highlighting. The gray header of the tag editor now displays the higher level tag, usually the FORM tag that defines this particular Web Form.

   Note: If a control is defined using several lines of HTML code, when you select the control, the first line of the code is displayed in the gray header of the tag editor. The additional code appears below in the tag editor window.

To view the HTML code for all controls

1. With the Designer displayed, drag several HTML elements from the **Tool Palette** to the Designer surface.

   The editor displays the HTML code for each element as you drop them on the Designer surface.

2. Click anywhere on the Designer surface to deselect all controls.

   This displays the code for all the controls in the tag editor, with syntax highlighting.

To modify a control

1. Click anywhere on the Designer surface to deselect all controls.
2. Locate the tag that corresponds to the control you want to modify.
3. Modify the code, and the change is immediately reflected in the control on the Designer surface.
4. Save your project to make the modifications permanent.

To change editor properties

1. Choose **Tools** ▸ **Options** ▸ **HTML/ASP.NET Options**.
2. Change any code editor properties.

3 Click **OK**.

Your changes take effect immediately.

To zoom between contents of the form and the form container

1 To zoom out so that you can view the HTML form definition, click the left-hand blue arrow in the gray header of the tag editor.

Note: You can only use this feature when the cursor is somewhere in the tag editor, rather than on the Designer surface.

2 To zoom in so that you can view only the content within the FORM tags, click the right-hand blue arrow in the gray header of the tag editor.

Note: You can only use this feature when the cursor is somewhere in the tag editor, rather than on the Designer surface.

To close the Tag Editor

1 Choose **Tools** ▸ **Options** ▸ **HTML/ASP.NET Options**.

2 Uncheck the **Display Tag Editor** option.

3 Click **OK**.

Working with ASP.NET User Controls

User controls provide a way to reuse common user interface functionality across ASP.NET web applications. For example, you might create user control that encapsulates a login screen. You could then add the user control to any Web Form that requires the login screen functionality. For more information about user controls, click the link at the end of this topic.

To create an ASP.NET user control

1. Open an ASP.NET application.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **New ASP.NET Files** and double-click on **ASP.NET User Control**.

   A new .ascx file is added to the **Project Manager** and the empty page is displayed in the Designer.

   Optionally, rename the .ascx file by right-clicking it in the **Project Manager** and choosing **Rename**. Any associated files, such as the .pas or .resx files, are also renamed.
3. Design the page by adding controls, setting properties, and adding code to the code-behind .pas file as needed.
4. Save and compile the project.

To add an ASP.NET user control to a Web Form

1. Open the Web Form to which you want to add the user control. Make sure the Designer is displayed.
2. Choose **Insert** ▸ **Insert User Control** to display the **Insert User Control** dialog box.
3. Select a user control from the drop-down list or use the **Browse** button to navigate to a user control file (.ascx).
4. Click **OK** to add the user control to the Web Form.
5. Optionally, in the **Object Inspector**, provide a descriptive name for the user control button with the **Id** property.
6. Save and compile the project.

The Web Form is displayed in the browser and the user control button is replaced with its encapsulated controls.

Tip: The runtime appearance of the user control depends on the appearance of the encapsulated page and controls, not the position of the user control button. If you are adding multiple user controls to a page, run the application to ensure that the controls do not overlap each other.

Database Procedures

Adding a New Connection to the Data Explorer

You can add new connections to the **Data Explorer**, which persist as long as the connection object exists.

To add a new connection

1. Choose **View** ▸ **Data Explorer**.
   This displays the **Data Explorer**.
2. Select a provider from the tree list.
3. Right-click to display a pop-up menu.
4. Choose **Add New Connection**.
   This displays the **Add New Connection** dialog.
5. Enter the name of the new connection.
6. Click **OK**.

Tip: If you need to modify your new connection settings, right-click on your new connection and scroll down to **modify a connection**. A **Connection Editor** dialog appears. Enter your connection settings and click **OK**.

Adding Aggregate Values with DBWebAggregateControl

You can use DBWebAggregateControl to apply one of several standard aggregation functions to a data column. The control displays the aggregate value in a text box, which also support a linked caption.

To create and configure a DBWebAggregateControl

1. Create a new ASP.NET web application and add your database connection, data adapter, dataset, and DBWebDataSource component to the application..
2. Set the Active property of BdpDataAdapter to **True**.
3. Place a DBWebAggregateControl component on the **Web Form Designer**.
4. Set the DBDataSource property of the DBWebAggregateControl to your DBWebDataSource1, which is the default name of the DBWebDataSource component.
5. Set the TableName property.
6. Choose the AggregateType property value from the drop down list.
7. Choose the ColumnName property from the drop down list.

   The text box is filled with the value based on the type of aggregate you selected and the values in the column you selected.

> **Note:** If you think there may be NULL values in your selected column, set the IgnoreNullValues property to **True**, otherwise you may get an error.

To set the caption for DBWebAggregateControl

1. In the **Object Inspector** enter the caption in the Caption property field.
2. Choose a position from the CaptionPosition property drop down list.

Adding an BDP Reconcile Error dialog to your BDP Application

You can modify your BDP applications to call the BDP Reconcile Error dialog to handle an update exception (as occurs sometimes when two people are trying to simultaneously update the same row of a database table).

To add a BDP Reconcile Error dialog:

1. Add a BDPDataAdapter component to your existing WinForm.
2. Choose the Events tab on the Object Inspector window
3. Double-click in the content section of the blank pull-down list next to the OnUpdateError event. This will populate the first level of the pull-down list. It will also create the code for the BdpDataAdapter method definition and implementation.
4. Add the lines that are in bold below to the method implementation to handle the event (the following example is using the C# language):

```
private void bdpDataAdapter1_OnUpdateError(object sender, Borland.Data.
Common.BdpUpdateErrorEventArgs e)
        {
         Borland.Data.Common.ReconcileErrorForm f = new Borland.Data.
Provider.ReconcileErrorForm( e );
            f.ShowDialog();
        }
```

5. Save the changes to your WinForm.

The BDP Reconcile Error dialog will now appear whenever one user is trying to modify data in the same row of a database that another user is working on. The dialog works as follows. As each row in a table is updated

Your new Error Reconcile Form will display four columns in the upper portion of the window, and six radio buttons in the bottom portion of the window. The following table describes each of the columns.

Column Label	Meaning
Column Name	The names of the columns of the table in which an error has occurred.
Current Row	The contents of the row that is currently in contention.
Original Row	What the row contained before the contentious data was entered.
Server Row	The last update that was saved to the Server. (This represents what the row contains on the server.)

The three radio buttons on the lower left portion of the window allow you to indicate how to continue processing after handling the error. You can only choose one option from the following three choices.

Radio Button Label	Meaning
Retry update using primary key	The error will be cleared, and then the update will be attempted again with the primary key. If the data row from the server cannot be found, this option will be disabled.
Skip current row and continue	Choose this option when you have decided not to attempt to update changes for the current row, but you want to try to update the rest of the rows.
Abort updates	The latest updates will not be applied, and error will be cleared, but no more updates will be attempted.

The three radio buttons in the lower right portion of the window allow you to indicate which data to write to the database. You can only choose one option from the following three choices.

Radio Button Label	Meaning
Use original values	Place the data from the Original Row column (described previously) into the row where the contention occurred.
Use server values	Place the data from the Server Row column, (described previously) into the row where the contention occurred.
Use current values	Place the data from the Current Row column, (described previously) into the row where the contention occurred.

Binding Columns in the DBWebGrid

There may be times when you want to modify the order in which columns appear in a DBWebGrid control. You can accomplish this task by binding columns manually, from within the **Property Builder**.

To open the Property Builder

1. Start a new ASP.NET application.
2. Add a data provider.
3. Add a DBWebDataSource object and connect it to a generated dataset.
4. Add a DBWebGrid control to your Web form.
5. Click the **Property Builder Designer verb**, located at the bottom of the **Object Inspector**.

   This displays the **Property Builder**.

To change column order

1. On the **Property Builder**, click the **General** tab.
2. Set the **DataSource** to the DBWebDataSource, or to the dataset the DBWebDataSource points to.
3. Click the **Columns** tab.
4. Select the columns you want to appear in the **Available Columns** list.
5. Click the right-arrow button to add the columns to the **Selected Columns** list.
6. Rearrange the column order, if you like, in the **Selected Columns** list.
7. You can change the column heading name as it appears in the grid by changing the **Header** text.
8. Click **Apply**.
9. Click **OK**.

Warning: If you choose to bind columns in this way, you must set the AutoGenerateColumns property to **False**. Setting this property to **True** raises a runtime error, and does not allow the visible restriction of columns at designtime. If the same column is bound to a grid more than once, you may get a runtime error.

Browsing a Database in the Data Explorer

Once you have a live connection, you can use the **Data Explorer** to browse database objects.

To browse database objects

1. Choose **View** ▸ **Data Explorer**.
2. Expand a provider node to expose the list of available connections.
3. Expand a connection node to view the list of database objects (tables, views, and procedures).

> **Note:** If you receive an error because your connection is not live, you should refresh your provider, and/or modify your connection.

To retrieve data from the database

1. Expand a connection in the **Data Explorer**.
2. Double-click a table name or view name to retrieve data.

   This operation returns a result set into a tabbed **Data Explorer** page in the **Code Editor**.

> **Tip:** You can also select a table in the **Data Explorer** and right-click to display a pop-up menu with a **Retrieve Data From Table** command.

To run a stored procedure

1. Choose **View** ▸ **Data Explorer**.
2. Expand a connection in the **Data Explorer** and locate a stored procedure.
3. Double-click the stored procedure to view its parameters.

   The parameters open in a separate page on the design surface.
4. Edit input parameters as necessary.
5. Click the Execute button in the top left corner of the page to execute the procedure.

   The result set appears in a datagrid.

> **Tip:** You can also select a procedure in the **Data Explorer** and right-click to display a pop-up menu with an **Execute** command.

Building a Database Application that Resolves to Multiple Tables

Developer Studio 2006 supports multi-table resolution with BDP.NET. Specifically, the DataSync and DataHub components are designed to provide and resolve a .NET DataSet from multiple heterogeneous data sources. In addition, these components support the display of live data at designtime, and provide and resolve master-detail data by generating optimal SQL for resolving to BDP data sources.

The DataHub acts as a conduit between a DataSet and a DataSync. The DataPort property for a DataHub can be set to any IDataProvider implementation. DataSync implements IDataProvider and has a Providers collection that can contain any .NET data provider that implements IDbDataAdapter. The GetData method for DataSync iterates through all the DataProviders in the collection and returns a DataSet. SaveData resolves DataSet changes back to the database through the DataProvider collection. While resolving changes through a BdpDataAdapter the resolver generates optimal SQL. For non-BDP data providers, their respective CommandBuilder is used.

Building a database application that resolves multiple tables consists of the following steps:

1. Create a simple database project from the **Data Explorer** with multiple BdpDataAdapter objects to connect to multiple providers
2. Add and configure a DataSync component to connect the providers
3. Add and configure a DataHub component to connect the DataSync to a DataSet

To create a database project from the Data Explorer

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.
2. Choose **View** ▸ **Data Explorer** to access the **Data Explorer**.
3. Expand the **Data Explorer Tree** to expose the providers and database tables you want to use.

   You must have a live connection to expand provider nodes. If you do not have a live connection, you may need to modify the connection string.
4. Drag and drop tables from one or more providers onto your form.

   For each table you drag onto your form, a BdpConnection and a BdpDataAdapter appear in the component tray.

   If you add multiple tables from the same provider, you can delete all but one BdpConnection for that provider.
5. Configure each BdpDataAdapter component.

   There is no need to set the Active or DataSet properties, as the DataSet will be populated by the DataHub component.
6. Add a DataSet component to your form from the **Data Components** category of the **Tool Palette**.
7. Add and configure a DataGrid component to your form from the **Data Controls** category of the **Tool Palette**.

   Set the DataSource property for the DataGrid to the name of the added DataSet component (for example, dataSet1).

To add and configure a DataSync component

1. Drag a DataSync component onto your form from the **Borland Data Provider** category of the **Tool Palette**.
2. In the **Component Tray**, select the DataSync component.
3. In the **Object Inspector**, select the Providers property, and click the ellipsis button to open the **DataProvider Collection Editor**.
4. In the the **DataProvider Collection Editor**, add a DataProvider for each table you want to provide and resolve.

You should have a DataProvider for each BdpDataAdapter in your project.

5 For each DataProvider, select the DataProvider in the Members pane, and set the DataAdapter property to the appropriate BdpDataAdapter.

6 When you have finished configuring your DataProviders, click OK to close the **DataProvider Collection Editor**.

7 In the **Object Inspector**, set the CommitBehavior property to specify how failures are handled during resolving.

There are three options for resolving logic:

- Atomic—transactions are attempted for each provider. If a transaction fails, no further transactions are attempted, and all preceding transactions are rolled back. If there are no failed transactions, all transactions are committed.
- Individual—a transaction is attempted for a provider, and if it succeeds, it is committed. The next transaction is attempted, and if it succeeds, it is committed, and so on. If a transaction fails for a provider, that transaction is rolled back, and no further transactions are attempted.
- ForceIndividual—a transaction is attempted for a provider, and if it succeeds, it is committed. The next transaction is attempted, and if it succeeds, it is committed, and so on. If a transaction fails for a provider, that transaction is rolled back, and the next transaction is attempted.

To add and configure a DataHub component

1 Drag a DataHub component onto your form from the **Borland Data Provider** category of the **Tool Palette**.

2 In the **Component Tray**, select the DataHub component.

3 In the **Object Inspector**, set the DataPort property to the added DataSync component (for example, DataSync1).

4 Set the DataSet property to the added DataSet (for example, dataSet1)

5 Choose **Run** ▸ **Run**.

The application compiles and displays a Windows Form with a DataGrid.

Building a Distributed Database Application

Data remoting is fundamental to developing distributed database applications. The .NET remoting technology provides a flexible and extensible framework for interprocess communication. With .NET remoting you can interact with objects in different application domains, in different processes running on the same machine, or in different machines on a network.

Using the RemoteServer and RemoteConnection components, you can easily migrate a client/server application that uses DataHub and DataSync components to a multi-tier DataSet remoting application. RemoteServer implements IDataService and publishes itself as a singleton server activated object (SAO). On the client side, the RemoteConnection properties form the URL for connecting to the RemoteServer. Channel specifies the protocol to use (TCP/IP or HTTP), Port specifies the port on which the RemoteServer is listening for requests, and URI refers to the unique resource identifier for the RemoteServer.

Building a distributed application with data remoting components consists of the following steps:

- Build a server-side Windows Forms application with one or more connections to a BDP.NET data provider, a DataSync component to collect the connections and set the commit behavior, and a RemoteServer component to set the communication protocol and URI for communicating with clients
- Build a client-side Windows Forms application with RemoteConnection component with properties to specify the connection to the server-side application, a DataHub component for passing data to and from a DataSet, and a DataGrid to display the data

Note: The RemoteServer component is hosted in Windows Forms applications without adding any additional code manually.

To create the server-side application

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.

2. Choose **View** ▸ **Data Explorer** to access the **Data Explorer**, and expand the **Data Explorer Tree** to expose the providers and database tables you want to use.

   You must have a live connection to expand provider nodes. If you do not have a live connection, you may need to modify the connection string.

3. Drag and drop tables from one or more providers onto your form.

   For each table you drag onto your form, a BdpConnection and a BdpDataAdapter appear in the component tray. If you add multiple tables from the same provider, you can delete all but one BdpConnection for that provider.

4. Configure each BdpDataAdapter component.

   There is no need to set the Active or DataSet properties, as the DataSet will be populated by the DataHub component on the client-side.

5. Drag a DataSync component onto your form from the **Borland Data Provider** category of the **Tool Palette**, and configure the following DataSync properties in the **Object Inspector**:

Property	Description
Providers	Specifies a collection of DataProviders to use as data sources. Click the ellipsis button to open the **DataProvider Collection Editor**, and add a DataProvider for each table you want to provide and resolve.
CommitBehavior	Specifies the logic (Atomic, Individual, or ForceIndividual) for handling failures during resolving.

6. Drag a RemoteServer component onto your form from the **Borland Data Provider** category of the **Tool Palette**, and configure the following RemoteServer properties in the **Object Inspector**:

Property	Description
DataSync	Specifies the DataSync that needs remoting. Select the DataSync from the drop-down list in the **Object Inspector**.
AutoStart	Specifies whether or not to start the remote server automatically when the application runs. Set this property to **True**.
ChannelType	Specifies the channel type: Http (HTTP) or Tcp (TCP/IP). Select the channel type from the drop-down list in the **Object Inspector**.
Port	Specifies the port the remote server will be listening on. Enter a new value, or accept the default port value, **8000**.
URI	Specifies the universal resource identifier for the remote server. By default, the URI property is the same as the Name property.

7 Choose **Run** ▸ **Run** to start the server-side application.

To create the client-side application

1 Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

The Windows Forms designer appears.

2 Drag a DataSet component onto your form from the **Data Components** category of the **Tool Palette**.

3 Drag a DataGrid component to your form from the **Data Controls** category of the **Tool Palette**, and set the DataSource property for the DataGrid to the name of the added DataSet component (for example, dataSet1).

4 Drag a RemoteConnection component onto your form from the **Borland Data Provider** category of the **Tool Palette**, and configure the following RemoteConnection properties in the **Object Inspector**:

Property	Description
ProviderType	Specifies the type of provider published by the remote server. In this case, the property should be set to **Borland.Data.Provider.DataSync**. If the remote server is running, you can select this value from the drop-down list. Otherwise, you must enter the value.
ChannelType	Specifies the channel type: Http (HTTP) or Tcp (TCP/IP). Select the channel type from the drop-down list in the **Object Inspector**. This should match the setting for the remote server.
Host	The name or IP address of the remote server.
Port	Specifies the port the remote server will be listening on. Enter a new value, or accept the default port value, **8000**. This should match the setting for the remote server.
URI	Specifies the universal resource identifier for the remote server. This should match the URI property for the RemoteServer component in the remote server application.

5 Drag a DataHub component onto your form from the **Borland Data Provider** category of the **Tool Palette**, and configure the following DataHub properties in the **Object Inspector**:

Property	Description
DataPort	Specifies the data source. Set the DataPort property to the added RemoteConnection component (for example, RemoteConnection1).
DataSet	Specifies the DataSet to hold the data retrieved from the specified data source. Set this property to the added DataSet (for example, dataSet1).

6 Choose **Run** ▸ **Run**.

The application compiles and displays a Windows Form with DataGrid.

Building a Windows Forms Database Application

The following procedure describes the minimum number of steps required to build a simple ADO.NET application using Windows Forms and BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of Windows Forms, ADO.NET, and BDP.NET at work. To instantiate and configure a data provider, you can also drag and drop objects from the **Data Explorer**, which is a tabbed window on the right-hand side of the IDE.

Building a BDP.NET project consists of three major steps:

1. Configure BDP.NET connection components and a data source.
2. Create and configure a BdpDataAdapter.
3. Connect a DataGrid to the connection components.

To configure connection components and a data source

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.
2. Drag and drop a BdpConnection component onto the **Designer**.

   The BdpDataAdapter, BdpConnection, and other BDP.NET components can be found on the **Tool Palette** in the **Borland Data Provider** area.
3. At the bottom of the **Object Inspector**, click the **Designer Verb Connection Editor**.

   Note: Designer verbs are action phrases that appear in the lower left-hand corner of the **Object Inspector**. When you move the cursor over the phrase, the cursor changes to a hand pointer.
4. Click **Add** to add a new connection.
5. Choose a provider type from the **Provider Name** drop down list box.
6. Type the name of the provider.
7. Click **OK**.
8. Set up the connection.
9. Click **OK**.

Tip: Alternatively, use Data Explorer to drag and drop a table on to the designer surface. Data Explorer sets the connection string automatically.

To set up a connection

1. Click the **Connections Editor Designer Verb** at the bottom of the **Object Inspector**.
2. In the **Borland Data Provider: Connections Editor** dialog box, select an existing connection from the **Connections** list or add a new connection.
3. In **Connection Settings**, enter the **Database** path.

Tip: If using Interbase, you would enter the path to your Interbase database, which may be located locally in `c:\Program Files\Common Files\Borland Shared\Data`. If connecting to a shared network location, you will need to enter the network path and you will need to have access rights for that remote server.

4 Complete the **UserName** and **Password** fields for the database as needed.

Tip: If you are using a sample Interbase database, the username and password are, respectively, `sysdba` and `masterkey`.

5 Click **Test** to confirm the connection.

A dialog appears indicating the status of the connection.

6 Click **OK**.

To create and configure a data adapter

1 From the **Tool Palette**, drag and drop a BdpDataAdapter component onto the **Designer**.

2 In the **Object Inspector**, expand the **SelectCommand** property in the **Fill** area.

3 Select the connection object from the **Connection** property drop down list box.

4 Click the **Configure Data Adapter** designer verb.

This displays the **Data Adapter Configuration** editor.

5 On the **Command** tab, select a table from the **Tables** list.

6 Select one or more columns from the **Columns** list.

7 Click **Generate SQL**.

To create a dataset

1 To make sure you get the data you want, click the **Preview Data** tab on the **Data Adapter Configuration** editor.

2 Click **Refresh**.

Column and row data should appear. If they don't appear, it may be that you either do not have a live connection to a database or your SQL statement is incorrect.

3 Click the **DataSet** tab.

4 Click **New DataSet**.

5 Either accept the default name or enter a more descriptive name.

6 Click **OK**.

A new DataSet component appears in the **Component Tray** at the bottom of the IDE.

To connect a DataGrid to a DataSet

1 In the **Component Tray**, select the BdpDataAdapter.

2 In the **Live Data** area of the **Object Inspector**, set the **Active** property to **True**.

3 Drag and drop a DataGrid component from the **Data Controls** area of the **Tool Palette** onto the **Designer**. If necessary, select the DataGrid object.

4 In the **Object Inspector**, select the DataSource property drop-down from the **Data** area.

5 Select the DataSet component that you generated previously (the default is dataSet1).

6 In the **Object Inspector**, select the DataMember property drop-down.

7 Select the appropriate table.

   The DataGrid displays data from the DataSet.

8 Choose **Run** ▶ **Run**.

   The application compiles and displays a Windows Form with DataGrid.

While presenting a minimum number of steps required to build an ADO.NET project, the preceding procedure demonstrates the major components of the Windows Forms, ADO.NET, and BDP.NET architectures at work, including: connections, datasets, and adapters. The adapter connects to the physical data source by way of a connection, sending a command that reads data from the data source and populates a dataset. Once populated, a datagrid displays data from the dataset.

Alternatively, use the **Data Explorer** to create and manage database connections.

Building an Application with DB Web Controls

The following procedures describe the minimum number of steps required to build a simple ASP.NET database application using DB Web Controls and BDP.NET. After generating the required connection objects, the project displays data in a DBWebGrid with a DBWebNavigator. Additional information is provided for other common DB Web Controls.

Users should already be familiar with creating an ASP.NET project using BDP.NET.

Building the simple ASP.NET application with DB Web Controls and BDP.NET consists of three major steps:

1. Prepare an ASP.NET project with BDP.NET or other connection components.
2. Drag and drop a DBWebDataSource onto the Designer and set its DataSource property to a DataSet, DataView or DataTable.
3. Drag and drop a DBWebGrid and other control onto the Designer.

To prepare an ASP.NET project for DB Web Controls

1. Create an ASP.NET project.
2. Set up BDP.NET or other data access components, setting the DataSource property to an existing DataSet, DataView, or DataTable.

   Tip: For more information about setting up BDP.NET data access components, see the related procedure for building an ASP.NET database application. Instead of using a DataGrid and adding a `DataBind` call, in the following procedure you use DB Web Controls without a `DataBind` call.

To configure a DBWebDataSource

1. Place a DBWebDataSource component on the **Designer**.
2. In the **Object Inspector**, select the DataSource property.
3. Select an existing data source (by default, this is called dataSet1).

To configure DB Web Controls

1. Place a DBWebNavigator component on the **Designer**.
2. In the **Object Inspector**, select a data source in the DBDataSource property drop-down.
3. In the **Object Inspector**, select a DataTable from the TableName property drop-down.

   Tip: If no TableName is available, verify that the BdpDataAdapterActive property is set to **True**.

4. Place a DBWebGrid on the **Designer**.
5. In the **Object Inspector**, select the data source from the DBDataSource property drop-down.
6. In the **Object Inspector**, select a DataTable from the TableName property drop-down.

   The grid displays data.
7. Place other DB Web Controls as needed.
8. Set the values for `DBDataSource`, `TableName`, and other properties as appropriate.

Note: For data-aware Column Controls (such as DBWebTextBox, DBWebImage, DBWebMemo, and DBWebCalendar) additionally set the `ColumnName` property. For data-aware lookup controls (such as DBWebDropDownList, DBWebListBox, and DBWebRadioButtonList), also set the `LookupTableName`, the `DataTextField`, and the `DataValueField` properties.

9 Choose **Run** ▸ **Run**.

The application compiles and the HTTP server displays a Web Form with a DBWebGrid displaying data.

Tip: Dragging web components from the **Tool Palette** places them in an absolute position on an ASP.NET web form. Double-clicking components in the **Tool Palette** leaves them in ASP.NET flow layout. Flow layout is much easier to manage. For instance, controls in an absolute position on a web form can overwrite other controls if they change sizes at runtime. Overwriting might occur when you add rows to and remove rows from a grid control, making the grid control change size.

Building an ASP.NET Database Application

The following procedure describes the minimum number of steps required to build a simple ASP.NET database application using BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of ASP.NET, ADO.NET, and BDP.NET at work.

Building an ASP.NET application with BDP.NET components consists of four major steps:

1. Create an ASP.NET project.
2. Configure BDP.NET connection components and a data source.
3. Add a DataBind call.
4. Connect a DataGrid to the connection components.

Tip: For testing purposes, use the employee.gdb database included with Interbase, if included with your version of the product.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
   The **New ASP.NET Application** dialog appears.
2. In the **Name** field, enter the name of your project.
3. In the **Location** field, enter the project path.

   Tip: Most ASP.NET projects reside in the IIS directory: Inetpub\wwwroot.

To change Web server settings (optional)

1. In the **New ASP.NET Application** dialog, click **View Server Options**
   The dialog expands to show additional server options.
2. Set the various read and write attributes of the project as needed or accept the defaults.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.
   The Web Forms Designer appears.

To configure data components

1. Drag and drop a BdpDataAdapter component onto the Designer. If necessary, select BdpDataAdapter.
2. In **Object Inspector**, select **Configure Data Adapter**.
   The **Data Adapter Configuration** dialog appears.
3. If necessary, select the **Command** tab. From the **Connection** drop-down, select **New Connection**.

4 The **Borland Data Provider: Connections Editor** dialog appears.

Tip: Alternatively, use Data Explorer to drag and drop a table on to the Designer surface. Data Explorer sets the connection string automatically.

To set up a connection

1 In **Borland Data Provider: Connections Editor**, select the appropriate item from the **Connections** list.

2 In **Connection Settings**, enter the **Database** path.

Note: If referring to a database on the local disk, prepend the path with localhost:. If using Interbase, for example, you would enter the path to your Interbase database: localhost:C:\Program Files \Borland\Interbase\Examples\Database\employee.gdb (or whatever the actual path might be for your system).

3 Complete the **UserName** and **Password** fields for the database as needed.

4 Click **Test** to confirm the connection.

A dialog appears confirming the status of the connection.

5 Click **OK** to return to the **Borland Data Provider: Connections Editor** dialog.

6 Click **OK** to return to the **Data Adapter Configuration** dialog.

In the **Command** tab, the areas for **Tables** and **Columns** are updated with information from your connection.

To set a command

1 In the **Select** area, enter an SQL command.

Tip: For Interbase's employee.gdb database, you might enter select * from SALES, as an example.

2 Click the **Preview Data** tab.

3 Click **Refresh**.

Column and row data appear.

4 Click the **DataSet** tab.

5 Confirm that **New DataSet** is selected.

6 Click **OK**.

New components for DataSet and BdpConnection appear on the Designer.

7 Select BdpDataAdapter component.

8 In **Object Inspector**, select the Active property drop-down and set the value to **True**.

To connect a DataGrid to a DataSet

1 Drag and drop a DataGrid web control onto the Designer. If necessary, select DataGrid.

2 In **Object Inspector**, select the DataSource property drop-down. Select the DataSet component that you generated previously (the default is DataSet1).

3 In **Object Inspector**, select the DataMember property drop-down. Select the appropriate table.

The DataGrid displays data from the DataSet.

To add a DataBind call

1. Use the **Object Inspector** drop-down to select the Web Form (**WebForm1** is the default).
2. In **Object Inspector**, select the **Events** tab.
3. Set the Load event to **Page_Load**.
4. In **Object Inspector**, double-click **Page_Load**.

   The code-behind Designer appears, cursor in place between event handler brackets.
5. Code the DataBind call:

```
[C#]
this.dataGrid1.DataBind();
```

```
[Delphi]
Self.dataGrid1.DataBind();
```

   Note: If you are using data aware controls, for instance from a third-party provider, you may not need to code the DataBind call.
6. Choose **Run** ▸ **Run**.

   The application compiles and the HTTP server displays a Web Form with the datagrid.

While presenting a minimum number of steps required to build a database project, the preceding procedure demonstrates the major components of the ASP.NET, ADO.NET, and BDP.NET architectures at work, including: providers, datasets, and adapters. The adapter connects to the physical data source via a provider, sending a command that will read data from the data source and populate a dataset. Once populated, a datagrid displays data from the dataset.

Once created, use other BDP.NET designers to modify and maintain the components of your project.

Creating a Briefcase Application with DB Web Controls

You can use DB Web Controls, XML caching, and the BDP.NET data adapters to create server-side briefcase applications. You can only create this type of application when using user authentication, to guarantee that each user has a unique copy of the XML file.

To create a briefcase application

1. Create a BDP.NET application.
2. Add a DBWebDataSource control and link to the BDP DataSet.
3. Configure the DBWebDataSource control to generate XML and XSD files.
4. Configure the AutoUpdateCache and UseUniqueFileName properties.
5. Configure an OnApplyChangesRequest to call the BdpDataAdapterAutoUpdate method.
6. Run the application.

To configure the AutoUpdateCache and UseUniqueFileName properties

1. Build a standard ASP.NET database application using the BDP.NET components and the DBWebDataSource component.
2. Specify XML and XSD filenames for non-existent files in the DBWebDataSource component.

   Note: It is best to create these files in the project directory or in a subdirectory of the project directory, typically on your web server.

3. Set AutoUpdateCache to **True**.
4. Set UseUniqueFileName to **True**.
5. Select the **Events** tab for the DBWebDataSource component.
6. Double-click the OnApplyChangesRequest field to display the event handler in the **Code Editor**.
7. Add the following code:

```
BdpDataAdapter1.AutoUpdate;
```

8. Choose **Run** ▸ **Run**.

   The first time the application runs, it creates the XSD file using the server metadata.

The first time a user runs the application, the application retrieves data from the server. When the user changes data, thereafter, the application saves those changes to the server in a unique filename based on the username. If the user shuts down the application and runs it again at a later time, the application restores the user's specific data. At this point, the user can undo or modify the data. Anytime the OnApplyChangesRequest is called successfully, the application deletes the unique user files and creates new ones.

Warning: If the tables or columns accessed by the application are altered after the application has run, you must delete the XSD file to avoid a mismatch between the XSD file and the server metadata. Otherwise, you can experience runtime errors and unpredictable behavior.

Creating an XML File for DB Web Controls

You can use XML files as your data source, particularly if you want to prototype applications without reading from and writing to a database. First you must create the XML file. The DBWebDataSource control provides a powerful way to create the XML file based on real database data. This procedure assumes that you can create a connection to a live database containing the data you want to use.

To create and use an XML file

1. Create an ASP.NET application using DB Web Controls.
2. Specify the XML file as a data source for a new ASP.NET application.

To create an ASP.NET application using DBWeb Controls

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Create a database connection and data adapter using the BDP.NET controls or other data adapter controls.
3. Drag and drop a DBWebDataSource control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
4. In the XMLFileName property or in the XMLSchemaFile property, specify a new file name of a file that does not yet exist.
5. Generate a DataSet from the data adapter.
6. Set the DataSource property of the DBWebDataSource to dataSet1.
7. Set the Active property of the data adapter to **True**.
8. Choose **Run** ▸ **Run**.

   This runs the application but also creates the XML file or XSD file and fills it with data from the DataSet.

To specify the XML file as a data source for a new ASP.NET application

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Drag and drop a **DataSet** component onto the **Designer** from the **Data Components** area of the **Tool Palette**.
3. Drag and drop a DBWebDataSource control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
4. Specify the existing XML file name in the XMLFileName property of the DBWebDataSource control.

   Note: If you created an XSD file instead of an XML file, you specify the XSD file name in this step.
5. Specify the DataSet component in the DataSource property of the DBWebDataSource control.
6. Drag and drop a DBWebGrid control onto the **Designer** from the **DB Web** area of the **Tool Palette**.
7. Set the DBDataSource property of the DBWebGrid to the name of the DBWebDataSource
8. Choose **Run** ▸ **Run** to display the application.

   The application pulls data from the DataSet and XML file to fill the DBWebGrid.

Warning: It is possible for you to specify an existing XML file in the XMLFileName property of your DBWebDataSource along with an active BdpDataAdapter and its DataSet. You can run the application and the DBWeb controls will display the data from the XML file. However, this is not the intended use or behavior of the XML capabilities of the DBWebDataSource. Although your XML file data may display properly, the results of an update or any other operations on the data will be unpredictable.

Creating Database Projects from the Data Explorer

You can drag and drop data from the **Data Explorer** to any forms such as Windows Forms or Web Forms, and Global.asax files. to populate datasets and quickly build a database project. This allows you to automatically hook up database components to your project and eliminates the need to provide a connection string, which can be prone to errors if entered manually.

To create a database project from the Data Explorer

1. Make sure you have a live connection to a database.
2. From the **View** menu, select **Data Explorer**.
3. Choose **File** ▸ **New** ▸ **Other** and select a Delphi for .NET project.

   Typically, this will be either a Windows Form, a VCL Form, or an ASP.NET application.
4. Expand the **Data Explorer Tree** by drilling down to the **Table** or **View** level.

   If the connection to your database is live, the small red x will disappear when you expand the connection node for the database. If it's not live, you may need to modify the connection string.
5. Using the cursor, grab one of the tables named in the list.
6. Drag and drop the table object onto your form.

   A BdpConnection and a BdpDataAdapter appear in the component tray.
7. Specify the appropriate database properties for each database component.

   For instance, set the Active property to *True* if you want to be able to view data in your component at design time.

Note: A DataGrid will not appear automatically so make sure you drop a DataGrid component onto your form to appropriately display data, when necessary.

Creating Metadata for a DataSet

When you choose to use an XML file for a data source in an ASP.NET application using DB Web Controls, you may need to create the metadata to structure the XML data in your DataSet. If you chose to create an XML file without an XML schema file (.xsd), you need to manually create the metadata. This procedure assumes that you have already created an XML file containing data.

To set up the application

1. Choose **File** ▸ **New** ▸ **ASP.NET Web Application** for either Delphi for .NET or C#.
2. Drag and drop a DBWebDataSource control onto the form.
3. Drag and drop a DataSet component onto the form.
4. Click the ellipsis button (...) next to the XMLFileName property of the DBWebDataSource and locate your XML file.
5. Select the DataSet component in the **Component Tray**.
6. Click the **Tables (Collection)** property to display the **Tables Collection Editor**.

To create the metadata

1. Click **Add** to add a new table to the collection.

   For the sake of illustration, we'll use the following XML records.

```
<?xml version="1.0" standalone="yes"> /// XML Declaration
<NewSongs>

 /// <song> becomes the table name in your DataSet.
    <song>

    /// <songid> becomes Column1 in your DataSet.
        <songid>1001</songid>

 /// <title> becomes Column2 in your DataSet.
        <title>Mary Had a Little Lamb</title>
    </song>
    <song>
        <songid>1003</songid>
        <title>Twinkle, Twinkle Little Star</title>
    </song>
</NewSongs>
```

2. Change the **TableName** property to song.
3. Click the **Columns (Collection)** property to display the **Columns Collection Editor**.
4. Click **Add** to add a new column.
5. Change the **ColumnName** property to songid.
6. Click **Add** to add another new column.
7. Change the **ColumnName** property to title.
8. Click **Close** to close the **Columns Collection Editor**.
9. Click **Close** to close the **Tables Collection Editor**.

   You have now created the metadata to match the XML file data.

Creating Table Mappings

Using the TableMappings property, you can map columns between a data source and an in-memory dataset. This allows you to use different, often more descriptive names for your dataset columns. You can also map a column in a database table to a column in the dataset different from that which is selected by default. The TableMappings property also allows you to create a dataset that contains fewer or more columns than those retrieved from the database schema.

To create a table mapping

1. Create an application.
2. Add and configure database components.
3. Set the table mappings in the TableMappings dialog.

Note: This procedure assumes you are using BDP.NET database components.

To create an application

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.
2. Click the Data Explorer tab to display your data sources.
3. Expand the list and locate a live data source.
4. Drag-and-drop a table name onto your Windows Form to add a data source to your application.

   You should see two objects in the **Component Tray**: a BdpDataAdapter and a BdpConnection.

For more information about how to create database applications, refer to the additional ADO.NET and database topics in this Help system.

To configure the database components

1. Select the BdpDataAdapter icon in the **Component Tray**.
2. Click the **Configure Data Adapter** designer verb to open the Data Adapter Configuration dialog.
3. Select the **DataSet** tab.
4. Click the **New DataSet** radio button.
5. Click **OK**.

   This creates a new dataset and displays an icon for it in the **Component Tray**.

To set table mappings

1. Select the BdpDataAdapter icon in the **Component Tray**.
2. Double-click the **Collections** field for the **TableMappings** property in the **Object Inspector**.

   This displays the **TableMappings** dialog.
3. If you want to use an existing dataset as a model for the columns, check the **Use a dataset to suggest table and column names** check box.

   This provides you with a list of column names from an existing dataset based on the schema of that dataset. The column names are not linked to anything when you use this process.

4 If you checked the **Use a dataset to suggest table and column names** check box, you can choose the dataset from the DataSet drop down list.

5 Select the source table from the **Source table** drop down list.

If there is more than one table in the data source, their names appear in the drop down list.

6 If you chose to use a dataset to suggest table and column names, and that dataset contains more than one table, you can select the table you want to use from the **Dataset table** drop down list.

The column names from the source table and from the dataset should appear in the **Column mappings** grid. As they are displayed by default, they represent the mapping from source to dataset; in other words, the data adapter reads data from each column named on the left side of the grid and stores the data in the dataset column named in the corresponding field on the right side of the grid. You can change the names on either side by typing new names or by selecting different tables. This allows you to store queried data into different dataset columns than the ones created in the dataset by default.

7 If you want to modify a mapping, type a new name in the Dataset table column next to the target Source table column.

This results in the data from the Source table column being stored in the new dataset column.

> **Note:** If you want to reset the column names so that the dataset columns match the data source columns, you can click the **Reset** button.

To delete a mapping

1 Select the grid row that you want to delete.

2 Click **Delete**.

This will cause the query to ignore that column in the source table and to not fill the dataset column with any data.

Executing SQL in the Data Explorer

You can write, edit, and execute SQL in an **SQL Window**, which is available from within the **Data Explorer**.

To open a SQL Window

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click the connection and choose **SQL Window**.

   This opens a tabbed **SQL Window** in the **Code Editor**.

To execute SQL

1. Enter a valid SQL statement or stored procedure name in the multi-line text box at the top of the **SQL Window**.
2. Click **Execute SQL**.

   If the SQL statement or stored procedure is valid, the result set appears in the bottom pane of the **SQL Window**.

   Note: The SQL statement or stored procedure must operate against the current connection and its target database. You cannot execute SQL against a database to which you are not connected.

3. Click **Clear All SQL** to clear the SQL statement or stored procedure from the multi-line text box.

Handling Errors in Table Mapping

Whenever you perform any type of comparison function between a data source and an in-memory data representation, there is potential for error. Errors can occur when a data source and its corresponding dataset do not share uniform numbers of columns, or when column types in a data source do not correspond to the column types in the dataset. In addition, other, internal errors can occur for which there is no designtime workaround. You can use both the MissingMappingAction property and the MissingSchemaAction property to respond to errors in your table mapping operations. Use the MissingMappingAction when you want to specify how the adapter should respond when the mapping is missing. Use the MissingSchemaAction when you want to specify how the adapter should respond when it tries to write data to a column that isn't defined in the dataset.

To set the MissingMappingAction property

1. Once you have created a BdpDataAdapter and have set up your table mappings, click the drop down list next to the MissingMappingAction property in the Object Inspector.
2. Select *Passthrough* if you want the adapter to load the data source column data into a dataset column of the same name, or, if there is no corresponding dataset column, if you want the adapter to perform the action specified in the MissingSchemaAction property.
3. Select *Ignore* if you want to keep data from being loaded when data source columns are not properly mapped to dataset columns.

   This could occur if mapped columns are of incompatible data types, lengths, or have other errors.
4. Select *Error* if you want the adapter to raise an error that you can trap.

To set the MissingSchemaAction property

1. Select *Add* if you want the data source table or column added to the dataset and its schema.

   Setting the MissingMappingAction property to *Passthrough* and the MissingSchemaAction to *Add* results in a duplication of data source table and column names in the dataset.
2. Select *AddWithKey* if you want the data source table or column added to the dataset and its schema along with the table's or column's primary key information.
3. Select *Ignore* if you don't want a table or column added to the dataset, when that table or column aren't already represented in the dataset schema.

   Specify *Ignore* when you want the dataset loaded only with data explicitly specified in the table mappings. This may be necessary if your adapter calls a stored procedure or a user-defined SQL statement that returns more columns than are defined in the dataset.
4. Select *Error* if you want the adapter to raise an error that you can trap.

Migrating Data Between Databases

The DataExplorer makes it easy to migrate data from one database to another, and even between providers. The DataExplorer lets you quickly copy a table from one database and paste it into another database. Both the structure and the data for the table or tables is migrated.

Data migration is supported by the BdpCopyTable class, which is available as a designtime component from the **Tool Palette**. You can use this component to programmatically migrate data.

Note: The BdpCopyTable class does not copy foreign keys or dependent objects.

To migrate multiple tables

1. Choose **View** ▸ **Data Explorer**.
2. Right-click a provider type, such as Interbase, and choose **Migrate Data**.

   The **Data Explorer** page for data migration opens in the **Code Editor**. This data migration page lets you select one or more tables from a source provider connection and a destination connection to which the tables will be migrated.
3. Choose a connection from the Source Connection drop-down list box.

   The tables associated with this connection appear in the list box beneath the connection.
4. Choose a connection from the Destination Connection drop-down list box.

   The tables associated with this connection appear in the list box beneath the connection.
5. Select one or more tables to migrate from the list of tables associated with the source connection.

   To select consecutive tables, click the first table, press and hold down the SHIFT key, and then click the last table. To select nonconsecutive tables, press and hold down CTRL, and then click each table.
6. Click the Include (>) button to include these tables for migration to the destination connection.

   The selected tables appear in the list of tables for the destination connection. If one of the selected tables has the same name as a table in the destination connection, it cannot be migrated.
7. Click Migrate to copy the tables to the destination connection.

   The Data Migration page shows the progress as SQL types are mapped, tables are created, data is retrieved from the source connection, and data is populated in the new table in the destination connection. The result of each operation is reported for each table.
8. Right-click the Tables node in the destination provider and choose Refresh.

   Nodes for any new tables appear.
9. Double-click a new table node to confirm its structure and contents.

   The table opens in a page on the design surface.

To migrate a single table

1. Choose **View** ▸ **Data Explorer**.
2. Expand the Tables node in the source provider, and select the database table containing the data and structure you want to migrate.

   You must have a valid connection to expand the provider nodes.
3. Right-click the table you want to migrate and choose **Copy Table**.
4. Expand the Tables node of the provider into which you want to migrate the data.

5 Right-click any table and choose **Paste Table**.
   The New Table Name dialog box appears.
6 Enter a name for the new table and click OK.
7 Right-click the Tables node in the destination provider and choose Refresh.
   A node for the new table appears.
8 Double-click the new table node to confirm its structure and contents.
   The table opens in a page on the design surface.

Modifying Connections in the Data Explorer

You can modify connections in a variety of ways from the **Data Explorer**.

To modify connections

1. Choose **View** ▸ **Data Explorer**.
2. Select a provider.
3. Right-click to display a pop-up menu to view your options.

To refresh a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a provider.
3. Right-click to display a pop-up menu.
4. Choose **Refresh**.
   This operation reinitializes all connections defined for the selected provider.

To delete a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Delete Connection**.
   This displays a confirmation message that asks if you want to delete the connection.
5. Click **OK**.

To modify a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Modify Connection**.
   This displays the **Connections Editor** dialog.
5. Make changes to the appropriate values in the editor.
6. Click **OK**.

To close a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Close Connection**.

If the connection is open, this operation closes it.

Note: If the **Close Connection** command is disabled in the menu, the connection is not open.

To rename a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Rename Connection**.
   This displays **Rename Connection** dialog.
5. Enter a new name.
6. Click **OK**.
   The **Data Explorer** displays the connection with its new name.

Modifying Database Connections

The basic elements of a connection string tend to be the same from one database type to another. However, each database type supports slightly different connection string syntax. This topic addresses those differences.

To modify different types of database connections

1. Click on the **Data Explorer** tab in the IDE.
2. Select the database type of your choice.
3. Right-click to display the popup menu.
4. Choose **Modify Connection** to display the **Connections Editor**.

   The properties in the Connections Editor are organized into three categories: Connections, Options, and Provider Settings. The Connections options designate the database and authentication parameters. The Options area includes various database-specific database options, including transaction isolation types. The Provider Settings area specifies assemblies and the client libraries required to accomplish the connection to the given database.

> **Note:** All of the procedures in this topic assume that you already have installed a database client, server, or both, and that the database instance is running.

To modify an InterBase connection

1. Either enter the database name or navigate to the database on your local disk or a network drive, by clicking the ellipsis button to browse.

   The standard supplied databases are typically installed into `C:\Program Files\Common Files\Borland Shared\Data`.
2. Enter the password and username.

   By default, these are `masterkey` and `sysdba`, respectively.
3. Set the following options, if necessary.

   The default values are shown in the following table.

Option	Description	Default
CommitRetain	Commits the active transaction and retains the transaction context after a commit.	False
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
RoleName	If there is a role for you in the database, you can enter the rolename here. The role is generally an authentication alias, that combines your identify with your access rights.	myRole
ServerCharSet	Specifies the character set on the server.	—
SQLDialect	Sets or returns the SQL dialect used by the client.	3
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting	ReadCommitted

	in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	
WaitOnLocks	Specifies that a transaction wait for access if it encounters a lock conflict with another transaction.	False

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Interbase,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Interbase
VendorClient	gds32.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

> **Note:** If you are writing ASP.NET applications, and are running the ASP.NET Web Forms locally for testing purposes, you might need to modify the path statement that points to your database, to include the `localhost:` designation. For example, you would modify the path shown earlier in this topic as such: `localhost:C:\Program Files\Common Files\Borland Shared\Data\employee.gdb`.

Note: Your connection string should resemble something like

```
database=C:\Program Files\Common Files\Borland Shared\Data\EMPLOYEE.GDB;
assembly=Borland.Data.Interbase,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=gds32.dll;provider=Interbase;username=sysdba;password=masterkey
```

To modify an MS SQL Server connection

1 Enter the database name in the **Database** field of the **Connections Editor**.

For example, use one of the sample MS SQL Server databases, such as Pubs or Northwind. There is no need to add the file extension to the name.

2 Enter the hostname.

If you are using a local database server, enter `(local)` in this field.

3 If you are deferring to your OS authentication, set **OSAuthentication** to **True**.

4 If you are using database authentication, enter the password and username into the appropriate fields.

By default, the SQL Server database username is `sa`.

5 Change the database options if necessary.

The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False

QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

6 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Mssql,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	MSSQL
VendorClient	sqloledb.dll

7 Click **Test** to see if the connection works.

8 Click **OK** to save the connection string.

Note: If you are writing ASP.NET applications, and are running the ASP.NET Web Forms locally for testing purposes, you might need to modify the path statement that points to your database, to include the `localhost:` designation, prepended to the path.

Note: Your connection string should resemble something like

```
assembly=Borland.Data.Mssql,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=sqloledb.dll;osauthentication=True;database=Pubs;username=;hostname=(local);
password=;
provider=MSSQL
```

To modify a DB2 connection

1 Enter the path to the database.

2 Enter the password and username into the appropriate fields.

3 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Db2,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	DB2
VendorClient	db2cli.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

To modify an Oracle connection

1 Enter the path to the database.

2 If you are deferring to your OS authentication, set **OSAuthentication** to **True**.

This means that the system defers to your local system username and password to login to the database.

3 If you are using database authentication, enter the password and username into the appropriate fields.

For example, the typical Oracle username and password for the sample database is `SCOTT` and `TIGER`, respectively.

4 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

5 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Oracle,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Oracle
VendorClient	oci.dll

6 Click **Test** to see if the connection works.

7 Click **OK** to save the connection string.

To modify an MS Access connection

1 Either enter the database name or navigate to the database on your local disk or a network drive, by clicking the ellipsis button to browse.

If you have the Office Component Toolkit installed, you might find Northwind in `C:\Program Files\Office Component Toolpack\Data\Northwind.mdb`.

2 Enter the username and password.

By default, you can generally try `admin` for the username and leave the password field empty.

3 Set the following database options, if necessary.

The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Msacc,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	MSAccess
VendorClient	msjet40.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

Note: Your connection string should resemble something like

```
database=C:\Program Files\Office Component Toolpack\Data\Northwind.mdb;
assembly=Borland.Data.Msacc,Version=2.0.0.0,
Culture=neutral,PublicKeyToken=91d62ebb5b0d1b1b;
vendorclient=msjet40.dll;provider=MSAccess;username=admin;password=
```

To modify a Sybase connection

1 Enter the path to the database.

2 Enter the password and username into the appropriate fields.

3 Set the following database options, if necessary. The default values are shown in the following table.

Option	Description	Default
BlobSize	Specifies the upper limit of the size of any BLOB field.	1024
ClientAppName	Client application name set by the middle-tier application.	—

ClientHostName	Client host name set by the middle-tier application.	—
LoginPrompt	Determines if you want the user to be prompted for a login every time the application tries to connect to the database.	False
PacketSize	Specifies the number of bytes per network packet transferred from the database server to the client.	512
QuoteObjects	Specifies that table names, column names, and other objects should be quoted or otherwise delimited when included in a SQL statement. This is required for databases that allow spaces in names, such as MS Access.	False
TransactionIsolation	Shared locks are held while the data is being read to avoid dirty reads, but the data can be changed before the end of the transaction, resulting in non-repeatable reads or phantom data. This specifies the value for the BdpTransaction. IsolationLevel property.	ReadCommitted

4 You should be able to accept the defaults for the following Provider Settings:

Option	Default
Assembly	Borland.Data.Sybase,Version=*Current Product Version*,Culture=neutral,PublicKeyToken=*Token #*
Provider	Sybase
VendorClient	libct.dll

5 Click **Test** to see if the connection works.

6 Click **OK** to save the connection string.

Note: Your connection string should resemble something like

```
assembly=Borland.Data.Sybase,Version=2.0.0.0,Culture=neutral,
PublicKeyToken=91d62ebb5b0d1b1b;vendorclient=libct.dll;database=Pubs;
username=admin;hostname=host1;password=;provider=Sybase
```

Passing Parameters in a Database Application

The following procedures describe a simple application that allows you to pass a parameter value at runtime to a DataSet. Parameters allow you to create applications at design time without knowing specifically what data the user will enter at runtime. This example process assumes that you already have your sample Interbase Employee database set up and connected. For purposes of illustration, this example uses the default connector IBConn1, which is set to a standard location. Your database location may differ.

To pass a parameter

1. Create a data adapter and connection to the Interbase employee.gdb database.
2. Add a text box control, a button control, and a data grid control to your form.
3. Configure the data adapter.
4. To add a parameter to the data adapter.
5. Configure the data grid.
6. Add code to the button Click event..
7. Compile and run the application.

To create a data adapter and connection

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.
2. Click on the **Data Explorer** tab and drill down to find the IBConn1 connection under the Interbase node.
3. Drag and drop the EMPLOYEE table onto the Windows Form.

   This creates a BdpDataAdapter and BdpConnection and displays their icons in the **Component Tray**.
4. Select the data adapter icon, then click the **Configure Data Adapter** designer verb in the **Designer Verb** area at the bottom of the Object Inspector.

   This displays the **Data Adapter Configuration** dialog.
5. Rewrite the SQL statement that is displayed in the Select tab of the dialog to:

   ```
   SELECT EMP_NO, FIRST_NAME, LAST_NAME, SALARY FROM EMPLOYEE WHERE FIRST_NAME = ?;
   ```

   As you can see, this statement is limiting the number of fields. It also contains a **?** character as part of the Where clause. The **?** character is a wildcard that represents the parameter value that your application passes in at runtime. There are at least two reasons for using a parameter in this way. The first reason is to make the application capable of retrieving numerous instances of the data in the selected columns, while using a different value to satisfy the condition. The second reason is that you may not know the actual values at design time. You can imagine how limited the application might be if we retrieved only data where `FIRST_NAME = 'Bob'`.
6. Click the **DataSet** tab.
7. Click **New DataSet**.
8. Click **OK**.

   This creates the DataSet that represents your query.

To add a parameter to the data adapter

1. Select the data adapter icon, then expand the properties under SelectCommand in the **Fill** area of the **Object Inspector**.

   You should be able to see your Select statement in the SelectCommand property drop down list box.
2. Change the ParameterCount property to 1.
3. Click the (Collection) entry next to the Parameters property.

   This displays the **BdpParameter Collection Editor**.
4. Click **Add** to add a new parameter.
5. Rename the parameter to *emp*.
6. Set BdpType to *String*, DbType to *Object*, Direction to *Input*, Source Column to *FIRST_NAME*, and ParameterName to *emp*.
7. Click **OK**.
8. In the **Object Inspector**, set the Active property under Live Data to **True**.

To add controls to the form

1. Drag and drop a TextBox control onto the form.
2. Drag and drop a Button onto the form.
3. Change the Text property of the button to *Get Info*.
4. Drag and drop a DataGrid data control onto the form.
5. Arrange the controls how you want them to appear, making sure that the DataGrid is long enough to display four fields of data.

To configure the data grid

1. Select the data grid.
2. Set the DataSource property to the name of the DataSet (dataSet1 by default).
3. Set the DataMember property to *Table1*.

   This should display the column names of the columns specified in the SQL statement that you entered into the data adapter.

To add code to the button Click event

1. Double-click the button to open the Code Editor.
2. In the button1_Click event code block, add the following code:

```
[C#]
bdpSelectCommand1.Close();
/* This closes the command to make sure that we will pass the parameter to */
/* the most current bdpSelectCommand.

            bdpDataAdapter1.Active = false;
/* This clears the data adapter so that we don't maintain old data

            bdpSelectCommand1.Parameters["emp"].Value = textBox1.Text;
/* This sets the parameter value to whatever value is in the text field.   */
```

```
            bdpDataAdapter1.Active = true;
/* This re-activates the data adapter so the refreshed data appears in the data grid. */
```

```
[Delphi]

Self.bdpSelectCommand1.Close();
/* This closes the command to make sure that we will pass the parameter to */
/* the most current bdpSelectCommand.

            Self.BdpDataAdapter1.Active := false;
/* This clears the data adapter so that we don't maintain old data

            Self.bdpSelectCommand1.Parameters['emp'].Value := textBox1.Text;
/* This sets the parameter value to whatever value is in the text field.   */

            Self.BdpDataAdapter1.Active := true;
/* This re-activates the data adapter so the refreshed data appears in the data grid. */
```

If you have changed the names of any of these items, you need to update these commands to reflect the new names.

3 Save your application.

To compile and run the application

1 Press Shift + F9 to compile the application.

2 Press F9 to run the application.

3 Type one of the names John, Robert, Roger, Kim, Terri, Katherine, or Ann into the text box.

4 Click the button.

This displays the employee number, first name, last name, and salary of the employee with that name in the data grid. If there is more than one person with the same first name, the grid displays all occurrences of employees with that name.

Using Standard DataSets

The standard DataSet provides an in-memory representation of one or more tables or views retrieved from a connected data source. Because of the level of indirection used in coding the underlying data structure, you are only able to see the column names from your data source at runtime. When you generate a DataSet, it retrieves everything you specified in your Select statement in the Data Adapter Configuration dialog. You can limit your columns by changing the Select statement and creating a new DataSet.

To use DataSets

1. Generate a DataSet.
2. Add multiple tables to a DataSet.
3. Define primary keys for DataTables in the DataSet.
4. Define column properties for your DataSet columns.
5. Define constraints for your columns.
6. Define relationships between tables in your DataSet.

To generate a DataSet

1. From the **Data Explorer**, select a data source.
2. Drill down in the tree, then drag and drop the name of a table onto your Windows Form or Web Form.

   This creates the BdpDataAdapter and BdpConnection for that data source and displays icons for those objects in the **Component Tray**.

   Note: You can also drag a data source only onto the form, rather than a table, but in that case, Developer Studio 2006 creates only a connection object for you. You must still create and configure the BdpDataAdapter object explicitly.

3. Click the BdpDataAdapter icon (named bdpDataAdapter1, by default) to select it.
4. Click the **Configure Data Adapter** designer verb in the **Designer Verb area** at the bottom of the **Object Inspector**.

   This displays the **Data Adapter Configuration** dialog.

5. If the SQL statement that is pre-filled on the dialog is acceptable, click the **DataSet** tab, otherwise, modify the SQL statement, then click the **DataSet** tab.
6. Select the **New DataSet** radio button.

   Tip: You can accept the default name or change the name of the DataSet.

7. Click **OK** to generate the DataSet.

   A DataSet icon appears in the **Component Tray** indicating that your DataSet has been created.

   Note: By reviewing the code for the DataSet in the Code Editor, you can see that the columns are defined as generic dataColumns, whose columnName properties are assigned the value of the column name from the database table. This differs from how a typed DataSet is constructed, wherein the object name is constructed from the actual database column name, rather than assigned as a property value.

To add multiple tables to one DataSet

1. From the **Data Explorer**, select a data source.
2. Drill down in the tree, then drag and drop the names of multiple tables, one at a time, onto your Windows Form or Web Form.

   This creates the BdpDataAdapter for each table and one BdpConnection for that data source and displays icons for those objects in the **Component Tray**.
3. Click the BdpDataAdapter icon (named bdpDataAdapter1, by default) to select it.
4. Click the **Configure Data Adapter** designer verb in the **Designer Verb area** at the bottom of the **Object Inspector**.

   This displays the **Data Adapter Configuration** dialog.
5. If the SQL statement that is pre-filled on the dialog is acceptable, click the **DataSet** tab, otherwise, modify the SQL statement, then click the **DataSet** tab.
6. Select the **New DataSet** radio button.

   Tip: You can accept the default name or change the name of the DataSet.
7. Click **OK** to generate the DataSet.

   A DataSet icon appears in the **Component Tray** indicating that your DataSet has been created.
8. Repeat the Data Adapter configuration for each of the other data adapters, but select **Existing Data Set** on the **DataSet** tab when generating the DataSets for all data adapters except the first one you configure.

   This generates a DataTable for each data adapter and stores them all in one DataSet.

   Note: It is also possible to generate multiple DataSets, either one for each data adapter, or combinations of DataTables.

To define primary keys for each DataTable in the DataSet

1. Select each data adapter in turn and set the Active property under **Live Data** in the **Object Inspector** to **True**.
2. Select the DataSet in the **Component Tray**.
3. In the **Object Inspector**, in the Tables property, click the ellipsis button.

   This displays the **Tables Collection Editor**. If you have set all of the data adapters' Active properties to **True**, the **Tables Collection Editor** will contain one member for each DataTable stored in the corresponding DataSet.
4. Select a table from the members list.
5. In the Primary Key field in the Table Properties, click on the DataColumn[] entry to display a pop-up list of column names.
6. Click the gray check box next to the column name of the column or columns that comprise the Primary Key.

   The number **1** appears in the gray check box when selected.
7. Define Column properties and Constraints for your Primary Key columns.

To define column properties for your DataSet columns

1. In the Tables Collection Editor, click the (Collections) entry next to Columns in the Table Properties pane.

   This displays the Columns Collection Editor for the selected column.
2. Set the property values for the individual columns.

3 Repeat the process for each column.

To define constraints for your columns

1 In the **Tables Collection Editor**, click the (Collections) entry next to Constraints in the **Table Properties** pane.

This displays the Constraints Collection Editor for the selected column.

2 Click **Add** to add either a Unique Constraint or a Primary Key Constraint.

3 If you selected Unique Constraint, the Unique Constraint dialog appears. Select one or more of the displayed column names. You can also select the Primary Key check box if you want to set the column as a primary key.

By setting the Unique Constraint on a column, you are enforcing the rule that all values in the column must be unique. This is useful for columns that contain identification numbers, such as employee numbers, social security numbers, part numbers, and so on.

Note: If you have already defined a primary-foreign key relationship between two tables, you may not be able to set a column as a primary key, based on the fact that it may already be set as the primary key, or based on a conflict with another relationship.

4 If you selected Foreign Key Constraint, the Foreign Key Constraint dialog appears. Select the tables you want to relate by choosing them from the Parent table and Child table drop down lists.

5 Click Key Columns to select the primary key column from the list.

6 Click Foreign Key Columns to select the foreign key column from the list.

Warning: The primary key and foreign key columns must have the same data type and must contain unique values. Columns that can contain duplicates are not good choices for primary or foreign keys. It is common to choose the same column name from each table for your primary-foreign key relationship.

To define relationships between tables in the DataSet

1 Once you have defined primary keys for each DataTable, select the DataSet in the **Component Tray** if it is not already selected.

2 Click the ellipsis button next to the Relations property in the **Object Inspector**.

This displays the blank **Relations Collection Editor** dialog.

3 Click **Add**.

This displays the **Relation editor** dialog

4 From the Parent table and Child table dropdown lists, choose the tables you want to relate.

5 Click the Key Columns field to choose a Primary Key column from the list of column names from the parent table.

6 Click the Foreign Key Columns field to choose a Foreign Key column from the list of column names from the child table.

Note: If you have already performed this procedure while setting constraints for your DataTables, you may find that all of the appropriate values are already established.

Warning: The primary key and foreign key columns must have the same data type and must contain unique values. Columns that can contain duplicates are not good choices for primary or

foreign keys. It is common to choose the same column name from each table for your primary-foreign key relationship.

7 Click **OK**.

8 Repeat the process to define additional relations between the same DataTables.

Using the Command Text Editor

In order to create a DataSet, your BdpDataAdapter needs to have at least a SQL Select statement defined for the CommandText property. This statement, once built, appears as the CommandText of the BdpCommand object for the BdpDataAdapter. You can enter this Select statement manually, or you can use the **Command Text Editor** to construct the statement, along with Update, Insert, and Delete statements, using a simple point-and-click mechanism. Using this method, once you have a connection to a live data source, you will be able to see the names of tables and columns in the **Command Text Editor**. You can pick from listboxes to build the statement. Also, if you create your BdpDataAdapter using the **Data Explorer** and a live connection to a data source, a boilerplate Select statement is created for you in the form `select * from tablename`. You can use this statement to return all rows from the named data source, or you can modify the statement prior to generating the DataSet.

To generate the commands

1. Select a connection from the **Connection** drop-down list box. This must be a BdpConnection you have already defined. Your associated BdpDataAdapter object must also be defined and must have the DataSet Active property set to **True**.

   This populates the Tables and Columns list boxes with data from the database.
2. Select a table from the Tables list box.
3. Select each column that you want to appear in your SQL statements.

   As you select the column names, they appear in the SQL text box.
4. Select the check box next to each statement type you want to generate.
5. Click the **Generate SQL** button.

Using the Connection Editor Designer

Each connection object can support multiple named connections. These connections can represent connections to multiple databases and database types.

To add a new connection

1. Select an existing BdpConnection component in the designer, or drop a BdpConnection component onto the designer to create a new object.
2. Click the component designer tab at the bottom of the **Object Inspector** to display the **Connection Editor** dialog.
3. Click **Add** to display the **Add New Connection** dialog.
4. Select a provider from the **Provider Name** drop-down list box.
5. Enter a new name for the connection in the **Connection Name** text box.
6. Click **OK**.
7. Enter the appropriate values for your particular data source.
8. Click **OK**.

To remove a connection

1. Select the connection type until it is highlighted.
2. Click **Remove**.

   A **Confirm Delete** dialog box appears.
3. Click **Yes**.

To rename a connection

1. Right-click on the connection and choose **Rename**.
2. Type the new name of the connection.
3. Click **OK**.

Using the Data Adapter Designer

The Data Adapter contains, at a minimum, a SQL Select statement of the SELECT command property. You can enter this statement yourself, or using the **Data Adapter** designer you can construct the Select, along with the Update, Insert, and Delete statements. The BdpCommandBuilder constructs the Update, Insert, and Delete statements based on the tables and columns you have selected. The **Data Adapter** designer uses a live connection to retrieve metadata from which you can build the appropriate SQL statements for manipulating the data you want to move from a DataSet back into your database.

To invoke the commands

1. Select a connection from the **Connection** drop-down list box. This must be a BdpConnection you have already defined.

   This populates the Tables and Columns list boxes with data from the database.
2. Select a table from the Tables list box.
3. Select each column that you want to appear in your SQL statements.
4. Select the check box next to each statement type you want to generate.
5. Click the **Generate SQL** button.
6. Edit the generated text if desired, or reselect different columns and click **Generate SQL** again.
7. Click **OK**.

> **Note:** Command components are automatically created as needed based on the selections in the dialog.

Using the Data Adapter Preview

Borland Developer Studio 2006 provides a tool that enables communication between a data source and a dataset. You can use the **Data Adapter Preview** to specify what data to move into and out of the dataset either in the form of SQL statements or stored procedures that are invoked to read or write a database.

To use the Data Adapter Preview

1. After you have dropped a BdpDataAdapter component onto the designer, click the **Configure Data Adapter** designer verb that appears at the bottom of the **Object Inspector**.
2. Click the **Preview** tab to display the **Data Adapter Preview**.
3. To limit the number of rows fetched, click the **Limit rows** check box.
4. Enter the number of rows you want the result set to contain, in the **Rows to fetch** text box.
5. Click **Refresh** to re-execute the query and to refill the list box with the new number of rows.

Using the DB Web Control Wizard

The **DB Web Control Wizard** helps you create a data-aware web control based on a standard web control.

To start the DB Web Control Wizard

1 Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **DB Web Control Library**.

> **Note:** You can also use the separate **DB Web Control Wizard** for C#. It works identically to the wizard described here.

This displays the **New DB Web Control Wizard**.

2 Enter a name for the control in the **Control Name** textbox.

3 Select **Bind to DataTable**.

This informs the wizard to add to the control file code that implements IDBWebDataLink. This interface defines the means to access data source and table information.

4 Select **Bind to DataColumn** if you want to bind to a column, for instance, if your control supports a single type of data.

This informs the wizard to add to the control file code that implements IDBWebColumnLink. This interface defines the means to access a column in the table accessed by way of IDBWebDataLink.

5 If you select **Bind to DataColumn** and your control is one of the lookup controls, such as a listbox, radio button group, or check box control, and you want the new control to be a lookup control also, check the **Supports Lookup** check box.

This informs the wizard to add to the control file code that implements IDBWebLookupColumnLink. This interface defines the means to access the lookup table, the text field and value field of the column accessed by way of IDBWebColumnLink.

The **DB Web Control Wizard** creates a template file and displays it in the **Code Editor**. You then modify this file to inherit from a specific DB Web control.

Using Typed DataSets

Typed DataSets provide certain advantages over standard DataSets. For one thing, they are derived from an XML hierarchy of the target database table. The XML file containing the DataSet description allows the system to provide extensive code-completion capabilities not available when using standard DataSets. Strong typing of DataSet methods, properties, and events allows compile-time type checking, and can provide a performance improvement in some applications.

To create a strongly typed DataSet

1. From the Database Explorer, select the data source you want to use.
2. Drag and drop the name of the database table you want to use onto your form.

   This displays a BdpConnection icon and a BdpDataAdapter icon in the **Component Tray**.
3. Select the BdpDataAdapter.
4. Click the **Configure Data Adapter** designer verb in the **Designer Verb** area beneath the **Object Inspector**.

   This displays the **Data Adapter Configuration** dialog.
5. Modify the pre-filled SQL statement if you like.
6. Click **OK**.

   Note: Do not create a DataSet by selecting the **DataSet tab** in the **Configure Data Adapter** dialog. That tab applies only to standard DataSets.
7. Click the **Generate Typed Dataset** designer verb in the **Designer Verb** area beneath the **Object Inspector**.

   This displays the **Generate Dataset** dialog.
8. Select the database table you want to use.
9. Click OK.

   This creates an instance of the typed DataSet and displays an icon *<DataSet Name>1* in the **Component Tray**. For example, if your DataSet is *DataSet1*, the new instance will be named *dataSet11*. You will also see that an XML .xsd file and a new program file appear in the **Project Manager** under your project.

To modify how columns appear

1. After you have created a new typed DataSet, drop a **DataGrid** component onto your form.
2. Set the DataSource property to point to the typed DataSet and the DataMember property to point to the target table.
3. Click the *(Collection)* entry next to the TableStyles property.

   This displays the **DataGridTableStyle Collection Editor**.
4. Click **Add** to add a new member to the members list.
5. Click the drop down list next to the MappingName property.
6. Click the *(Collection)* entry next to the GridColumnStyles property.

   This displays the **DataGridColumnStyle Collection Editor**.
7. Click **Add** to add a new item to the members list.

   Note: By default the item is created as a Text Box Column. You can also expand the **Add** button and select the BoolColumn if you want a boolean.

8 Click the MappingName property, select the column you want to display in your grid, then change any additional properties you want, including the header name that will appear as the column header in the runtime grid.

9 Click **OK** twice.

Note: When you build and run the application, only the columns that you explicitly defined by following the steps in this procedure appear.

To modify the structure of the dataset

1 In the **Project Manager**, double-click the *.xsd* file that contains the XML definition of your dataset.

2 Edit the XML file to reflect how you want the dataset to be structured.

You can change data types, names, and anything else about the structure.

3 If you have the program code file (<dataset>.cs or <dataset>.pas) open in the **Code Editor**, close it now.

4 Choose **Project** ▸ **Compile** to recompile the .xsd file.

If you re-open the program code file, you will see that the file contains the changes you made to the XML in the *.xsd* file.

To set the Namespace property for a dataset

1 In the **Project Manager**, double-click the *.xsd* file that contains the XML definition of your dataset.

2 Find the targetNamespace property.

3 Change the following text to a relevant namespace:

```
http://www.changeme.now/DataSet1.xsd
```

4 If you have the program code file (<dataset>.cs or <dataset>.pas) open in the **Code Editor**, close it now.

5 Choose **Project** ▸ **Compile** to recompile the .xsd file.

If you re-open the program code file, you will see that the InitClass() class now contains the new namespace.

ECO Framework Procedures

Adding a Derived Association End to an ECO Class Diagram

You can use OCL to derive association ends. Unlike the case of derived attributes, the expression must result in a single object or a collection of objects, depending on the multiplicity of the association end. A derived association end must be derived from another association on the class.

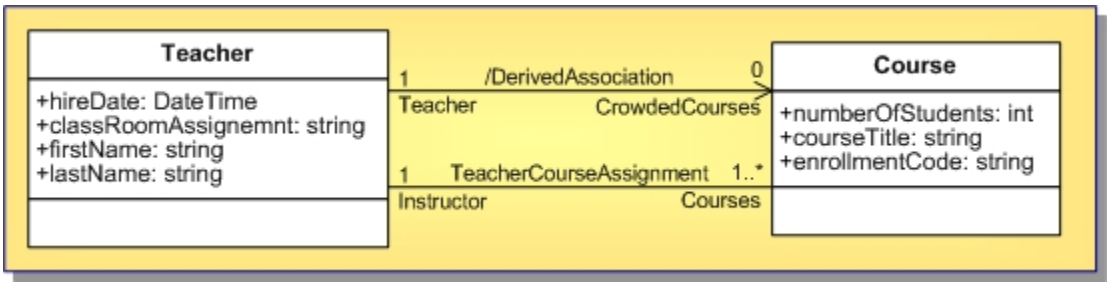
To add a derived association end

1. Add a new association between the two classes of interest.
2. Select the new association on the **ECO class diagram**.
3. Set the `derived` property of the new association to `true`.
4. Expand either (or both) of the `End1TaggedValues` or `End2TaggedValues` properties of the new association.

   Either or both ends of the association may be derived.
5. In the **Object Inspector**, for either the `End1TaggedValues` or `End2TaggedValues` property, click the property editor ellipses for the `Derivation OCL` property.

   The **OCL Expression Editor** will open.
6. Create the OCL expression for the derived association end.
7. Click OK to close the **OCL Expression Editor**.

The following diagram shows an example of a derived association end. The name of the derived association is `/DerivedAssociation`. The `Derivation OCL` property of the `CrowdedCourses` association end contains the OCL expression `self.Courses->select(numberOfStudents > 30)`. Notice how the OCL expression for the derived association end uses the existing `TeacherCourseAssignment` association. The semantics of the `CrowdedCourses` association end and its derivation OCL expression are that a `Teacher` object has an attribute called `CrowdedCourses`. The `CrowdedCourses` attribute gives the collection of all `Course` objects with more than thirty students enrolled.



Adding a Derived Attribute to an ECO Class

To create an attribute with a derived value, set the attribute's `derived` property to `true` in the **Object Inspector**. Note that the names of derived attributes are automatically prefixed with a slash ('/'). This is a UML convention. Also note that setting the `derived` property to `true` causes the persistence property to be set to transient, meaning the attribute will not be stored in the database.

To add a derived attribute

1 Right-click on the ECO class, and choose **Add** ▸ **Attribute**.

> **Note:** You may add an attribute either from the **ECO class diagram** context menu, or from the **Model View** context menu.

2 Set the attribute's name and type in the **Object Inspector**.

3 Set the attribute's `derived` property to `true` in the **Object Inspector**.

> **Note:** The IDE automatically adds a slash ('/') to the attribute name, and sets the `transient` property to `true`.

4 Click the `Derivation OCL` property editor ellipses.

The **OCL Expression editor** will open. Create an OCL expression that results in the same type as the derived attribute.

Adding a Guard Expression to a State Transition

Guard expressions are written in Object Constraint Language (OCL). On a state machine diagram guard expressions result in a boolean value that determines whether or not a transition can occur. If the result of the expression is `true`, the transition can occur.

To add a guard expression to a state transition

1. In the **Model View**, double-click the state machine diagram icon of the class you wish to work with.

   The state machine diagram for the class opens.
2. On the diagram, select the transition for which you want to add a guard.
3. In the **Object Inspector**, click **[...]** on the `guard` property.

   The **OCL Expression Editor** opens.
4. Use the **OCL Expression Editor** to construct an OCL expression that results in a boolean value.

   Using the Expression Editor to Build OCL and Expressions

Adding a PersistenceMapperClient to an ECO Space

To add a PersistenceMapperClient to an ECO space

1. Save and compile your project.
2. Open the ECO space source file for your project.
3. Select the **Design** tab of the ECO space source file.
4. Place a PersistenceMapperClient component onto the **ECO Space Designer**.
5. Select the PersistenceMapperClient component, and set its Url property to the URL of the persistence server, for example, `tcp://localhost:4243/PersistenceServer`.

   Note: The `PersistenceMapperProvider` code template creates a method called `RegisterTcpServer` that you can use as an example of registering a persistence server. The port and server name, shown in the URL above, are both set in the `RegisterTcpServer` method.

6. Click an empty part of the **ECO Space Designer**.

   The **Object Inspector** displays the properties of the ECO space itself.

7. Set the PersistenceMapper property of the ECO space to the PersistenceMapperClient component you created in step 4.

Adding a PersistenceMapperSharer to an ECO Space

To add PersistenceMapperSharer to an ECO space

1. Save and compile your project.
2. Open the ECO space source file for your project.
3. Select the **Design** tab of the ECO space source file.
4. Place a PersistenceMapperSharer component onto the **ECO Space Designer**.
5. Select the PersistenceMapperSharer component, and set its MapperProviderType property to the `PersistenceMapperProvider`.
6. Click an empty part of the **ECO Space Designer**.

   The **Object Inspector** displays the properties of the ECO space itself.
7. Set the PersistenceMapper property of the ECO space to the PersistenceMapperSharer component you created in step 4.

Adding a Reference to an ECO Package in a DLL

You can add a reference to ECO packages that are compiled in separate DLLs. Classes defined in the external package are read-only. You may derive new classes from them, and draw unidirectional associations to them, but you may not add attributes or methods to them.

Note: The **ECO Package in a DLL** wizard creates a project that builds a DLL containing an ECO package.

To add a reference to ECO packages in a DLL

1. Select the **Project Manager** window.
2. Right-click the executable node of the project that is to reference the DLL and choose **Referred ECO Packages**.

   The **Referred ECO Packages** dialog appears.
3. Click **Add**.

   The **Open file** dialog opens.
4. In the **Open file** dialog, navigate to the ECO package file that you wish to refer to in your project.

   The ECO package file is named <PackageName>.ecopkg, where PackageName is the name you gave to the ECO package in the IDE.
5. In the **Open file** dialog, select the package file and click **OK**.

   The selected package file will appear in the **Referred ECO Packages** dialog.

   Repeat this procedure, adding all the ECO package files you wish to refer to in your project.
6. In the **Project Manager**, right-click the executable node and choose **Add Reference**.

   The **Add Reference** dialog opens.
7. In the **Add Reference** dialog, click the **Browse** button and navigate to the DLL that contains the ECO package you added previously.

   Note: Repeat this step to add a reference to the DLL for each ECO package file you added.
8. In the **Add Reference** dialog, click the **Ok** button when you are done adding DLL references.

The referenced ECO packages will appear in the **Model View**. You can view the diagrams in the model and reference its classes and associations in your source code.

Adding a Region to a State

Each state on a state machine diagram contains at least one region. You may add additional regions, allowing for multiple, concurrent substates.

Multiple regions within a state are delineated by a dotted line.

You may add regions to a state using the **ECO state machine diagram**, or the **Model View**.

To add a region using the ECO state machine diagram

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Expand the ECO class node.
4. Double click on the state machine diagram icon to open the **ECO state machine diagram** for the class.

   The state machine diagram for the class opens.
5. Right-click within the state to which you want to add a new region.

   Note: Be sure to click outside of any existing regions, so that the selection handles appear on the state rectangle itself.
6. Choose **Add** ▸ **Region** from the context menu.

To add a region using the Model View

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Expand the ECO class node.
4. Expand the ECO state machine diagram node that contains the state to which you will add the region.
5. Right-click on the state and choose **Add** ▸ **Region**.

Both the **Model View**, and the **ECO state machine diagram** will be updated to show the new region.

To edit the properties of a region

1. Select the region.

   Note: You can select the region either in the **Model View**, or on the **ECO state machine diagram**.
2. Edit the properties of the region in the **Object Inspector**.

Adding a Trigger Method to an ECO Class

A trigger method may be added to a class using either the **Class Diagram** or the **Model View**.

Note: A trigger is an operation on a class that has its `Is Trigger` attribute set to `true`. You may convert any existing operation to a trigger by setting the `Is Trigger` attribute in the **Object Inspector**.

To add a trigger method using the Model View

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Right-click on the class and choose **Add** ▸ **Trigger** .

A new trigger method called Trigger_1 is added to the class. Edit the name of the method as necessary.

To add a trigger method using the ECO class diagram

1. Choose **View** ▸ **Model View**.
2. Double-click the ECO package icon that contains the class.
   The **ECO class diagram** displays.
3. Right-click on the class and choose **Add** ▸ **Trigger**.

Both the **Model View** and the **ECO class diagram** will be updated to show the new trigger.

To change the properties of a trigger

1. Click the trigger method, either on the **ECO class diagram** or in the **Model View**.
2. Edit the properties of the trigger method in the **Object Inspector**.

If your trigger method has parameters, you must edit them in the **Object Inspector** rather than in source code. This is so the parameters will not be overwritten if you regenerate ECO source code.

Adding an ECO Enabled Windows Form to a Project

You can add an ECO Enabled Windows Form to a project using the ECO Enabled Windows Form code template.

Note: The ECO wizards are available in both C# and Delphi for .NET projects. The functionality of the wizards is identical. The only difference is in the language used to generate source code files.

To add an ECO-enabled Windows form to your project

1. Open an ECO project.
2. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
3. Select the **New ECO Files** category.
4. Double-click **ECO Enabled Windows Form**.

The **ECO Enabled Windows Form** wizard adds a new `System.Windows.Forms.Form` to your project.

The new class provides a constructor that takes an instance of the ECO Space of the application. The constructor then uses this instance to initialize the ECO Space property of the form. An ECO-enabled form also provides a root handle and four extender providers, which are described in *Creating a New ECO Windows Forms Application*.

Adding an ECO UML Package to a Project

You can add an ECO UML package to a project using the **Model View** context menu.

To add a new ECO UML package to your project

1. Open an ECO project.
2. Choose **View** ▸ **Model View**.

   The **Model View** appears.
3. Right-click on the top-level node in the **Model View** and choose **Add** ▸ **ECO Package**.

> **Note:** To nest an ECO package within another one, right click on the ECO package node in the **Model View** and choose **Add** ▸ **ECO Package**.

The IDE creates a new source code file that contains the declarations necessary for the new UML package. The default name of the new package is `Package_N`, where `N` is an increasing integer.

You can populate the UML package with new classes and draw the relationships between them on the **ECO Class Diagram** surface. As you design the classes, the IDE generates source code into the source file that contains the ECO UML package. The generated source code contains the necessary ECO attributes to support the services provided by the ECO framework.

Adding an Effect to a State Transition

State transitions can have effects that occur when the trigger method of the transition is called. Transition effects are written in ECO Action Language.

To add an effect to a state transition

1. In the **Model View**, double-click the state machine diagram icon of the class you wish to work with.

   The state machine diagram for the class displays.
2. On the diagram, select the transition for which you want to add an effect.
3. In the **Object Inspector**, click **[...]** on the `Effect` property.

   The **ECO Action Editor** displays.
4. Use the **ECO Action Editor** to construct an ECO Action Language expression.

   Using the Expression Editor to Build OCL and Expressions

Adding and Configuring a Connection Handle on an ECO Space

This procedure shows how to add a connection handle component to the **ECO space designer**.

To add and configure a connection handle

1. Locate the **Borland Data Provider** category in the **Tool Palette**.

   Note: If you are using a SQL Server persistence handle, locate the **Data Components** category.

2. Place a BdpConnection or SqlConnection component onto the **ECO Space Designer**.
3. Select the persistence mapper component on the designer surface.
4. Set the Connection property of the persistence mapper component to the connection handle you created in step 2.
5. Set the default vendor-specific configuration settings of the persistence mapper by right-clicking the persistence mapper component, and choosing the appropriate item from the context menu.

   For example, to set the default settings for an InterBase database, select **InterBase ▸ dialect 3 setup** from the context menu.

6. Right-click the connection handle component and choose **Connection Editor**.

   The ConnectionString will vary depending on the database vendor. For an InterBase database, you will need to edit the connection string to reflect the correct path to the database file. Default, vendor-specific connection strings are available both from the **Connections Editor** dialog box, and from the drop-down list of the ConnectionString property in the **Object Inspector**.

Adding Columns and Nestings to an ECO Handle

You can add columns to handles to accommodate computed or derived values that have no corresponding designtime field in a class or in an underlying data source. The ECO framework provides the ability to add columns to handles at designtime. GUI components, such as datagrids, that use the handle as a datasource will display the additional columns. This procedure assumes you have read the ECO and modeling overviews listed below.

To add a column to an expression handle

1. Open an ECO Windows Forms application with a datagrid and an expression handle already defined.
2. Select the expression handle in the **Component Tray** of the form designer.
3. Click **[...]** on the **Column** field in the **Object Inspector**.

   The **Column Collection Editor** appears.
4. Click **Add**.
5. Click **[...]** on the **Expression** field in the **Properties** pane.
6. Create the OCL expression by double-clicking objects in the right-hand text box of the **OCL Expression Editor** and adding elements.

   For example, double-click a class name, then double-click `allInstances` to add it to the expression.
7. Click **OK** to close the **OCL Expression Editor**.
8. Click **OK** to close the **Column Collection Editor**.
9. Compile the application.

   If you are using a datagrid that references the expression handle for which you created a new column, the new column will appear in the designtime datagrid.

If the column you are adding is a relationship to another class, the column is called a nesting. The OCL expression you enter for the column will return an object or objects that have their own set of attributes. You must configure the nested columns using the **Nesting Collection Editor**.

To add a nesting to a column

1. Select the expression handle component that contains a nested column.
2. Click **[...]** to open the **Column Collection Editor** for the expression handle.
3. In the **Column Collection Editor**, select the nested column.
4. Set the Nested property of the column to True.
5. Enter a name in the NestingName property.

   You will refer to this name in the **Nesting Collection Editor**.
6. Click **OK**.
7. Click **[...]** of the Nestings property.
8. Click **Add**.

   Enter the Name of the nesting; this is the name you entered in step 5, above.
9. Click **[...]** to open the **Column Collection Editor**.

   Each nesting has its own collection of columns, which you must configure using the **Column Collection Editor**.

   Note: The columns in a nesting might contain more nestings themselves.

Adding Entry and Exit Actions to a State

States can have entry and exit actions associated with them. These actions are performed on entry to a state, and when a trigger causes a state to be exited, respectively. Entry and exit actions are written using ECO Action Language, which is an extension of the Object Constraint language (OCL) that allows side-effects.

To add entry and exit actions to a state

1. In the **Model View**, double-click the state machine diagram icon of the class you wish to work with.

   The state machine diagram for the class displays.
2. On the diagram, select the state for which you want to add entry and exit actions.
3. In the **Object Inspector**, click **[...]** on the `Entry action` or on the `Exit action` property.

   The **ECO Action Editor** opens for the entry action, or the exit action, respectively.
4. Use the **ECO Action Editor** to construct an ECO Action Language expression.

   Using the Expression Editor to Build OCL and Expressions

Adding States and Substates to an ECO State Machine Diagram

You may add states to an **ECO state machine diagram** using either the **Model View**, or the state diagram context menu. You may also drag states onto the diagram from the **Tool Palette**.

A state contains one region by default. If you need to model sequential substates, you may add substates to a region within a state. When concurrent substates are required, you must add more regions to the state. With noted variations, you add substates to regions using the same procedures described below. The main difference is that substates are added to a region within a state, rather than at the top level of the diagram itself.

To add a state using the ECO state machine diagram context menu

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Expand the class node.
4. Double click the state machine diagram icon to open the **ECO state machine diagram** for the class.
5. Right-click on an empty part of the state machine diagram.

   Note: To add a substate, right-click within a region of the composite state instead of on an empty part of the state machine diagram.

6. Choose **Add** ▸ **State**.

To add a state using the Model View

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Expand the class node.
4. Right-click on the ECO state machine diagram icon and choose **Add** ▸ **State**.

   Note: To add a substate, expand the node of the composite state, and right-click on the region icon in the **Model View**.

To add a state using the Tool Palette

1. Choose **View** ▸ **Model View**.
2. Expand the ECO package that contains the class of interest.
3. Expand the class node.
4. Double click on the state machine diagram icon to open the **ECO state machine diagram** for the class.
5. In the **Tool Palette**, click the state icon and drag it onto the diagram.

   Note: To add a substate, drag the state to a region within the composite state.

You may add initial and final states by choosing **Add** ▸ **Initial** or **Add** ▸ **Final** from either the diagram context menu or the **Model View** context menu. Similarly, you may drag initial and final states from the **Tool Palette.**

A state machine may have only one initial state (regions may have their own initial states). If an initial state already exists, the **Initial** menu item will be disabled. When dragging an initial state from the **Tool Palette**, the IDE will not allow you to drop a second initial state onto the diagram.

To edit the properties of a state or substate

1 Select the state or substate.

> **Note:** You may select the state or substate either on the **ECO state machine diagram**, or in the **Model View**.

2 Edit the properties of the state or substate in the **Object Inspector**.

Building Applications with the ECO Framework

Building an ECO-enabled application consists of a number of steps, each with its own set of procedures. This topic presents an overview of the entire process. It is assumed you are familiar with basic ECO concepts; please see the links below for more information.

To create an ECO-enabled application

1 Create an ECO application using one of the following code templates in the **New Items** dialog box (File ▸ New ▸ Other):

- **ECO Windows Forms Application**
- **ECO ASP.NET Web Application**
- **ECO ASP.NET Web Service Application**

> **Note:** The code templates are available for both Delphi for .NET, and C# projects.

2 Create or edit your model using the integrated UML class diagramming tools:

- The **Model View** shows the logical view of your project, as opposed to the file-oriented view of the **Project Manager**.
- The **ECO Class diagram** surface allows you to draw your classes and build relationships between them.
- The **ECO State Machine diagram** allows you to model the behavior of the classes in your model.
- The **Tool Palette** contains the UML elements that you may add to the diagram.
- The **Object Inspector** allows you to edit properties of the elements in your model.

3 If you will be using a relational database, create a new, empty database using the vendor-supplied tools.

4 Configure the ECO Space of the application using the **ECO Space Designer**.

The ECO Space contains the model, and the objects created by your application as it runs; it is a middle layer between the user interface and the persistent storage of the application. The ECO Space design tab contains all the tools to:

- Configure the application's persistence settings (RDBMS or XML file).
- Create or evolve the database schema.
- Select the ECO UML packages that you wish to persists from the model.
- Validate the model.
- Reverse engineer and wrap an existing relational database with ECO source code.
- Upgrade an ECO project created with a prior version of the IDE.

5 Build a user interface for your application.

ECO applications are built using the **WinForms designer**; there is nothing different about ECO applications in that respect. You can connect data-aware .NET components on your forms to the objects in your ECO Space through element handles such as ExpressionHandle. The ExpressionHandle component provides a way to retrieve objects from the ECO Space using an OCL expression. The element handle components implement the interfaces required to render their values in a data-aware component. You can use the **OCL Expression Editor** to enrich the specification of your model by adding invariant constraints and derived attributes.

6 Evolve your application using ECO code templates in the **New Items** dialog box:

- **ECO Enabled Windows form**
- **ECO Space**

- **ECO PersistenceMapperProvider**

7 Deploy your ECO enabled application.

Configuring a PersistenceMapperMultiDb Component

A PersistenceMapperMultiDb component is used when data resides in more than one database. The ECO space must be configured with one PersistenceMapper for each database. Each PersistenceMapper must then be configured to connect to the correct database.

The PersistenceMapperMultiDb component manages the collection of PersistenceMapper components.

To configure a PersistenceMapperMultiDb component

1 Configure the ECO space with one PersistenceMapper component for each database.

> **Note:** Do not connect the ECO space to the PersistenceMapper components added in this step. The ECO space will be connected to the PersistenceMapperMultiDb component later in this procedure.

Configuring the Persistence Method of an ECO Space

2 Add a database connection component for each PersistenceMapper.

Adding and Configuring a Connection Handle on an ECO Space

3 If you are using a custom OR mapping file, you must configure a FileMappingProvider for each PersistenceMapper component.

Using a Custom Object-Relational Mapping File

4 In the **Tool Palette**, drag a PersistenceMapperMultiDb component from the **Enterprise Core Objects category** onto the **ECO space designer**.

5 In the **Object Inspector**, click **[...]** for the PersistenceMappers property.

The **PMapperDef Collection Editor** displays.

6 Use the **PMapperDef Collection Editor** to add each PersistenceMapper on the ECO space to the PersistenceMapperMultiDb component.

> **Note:** The `name` property is a logical name for the database. If you are using a custom OR mapping file, the logical name you give to each PersistenceMapper must correspond to the name in the OR mapping file.

7 Click in an empty region of the **ECO space designer**.

8 In the **Object Inspector**, set the PersistenceMapper property to the PersistenceMapperMultiDb component..

Configuring an OclVariables Component

This procedure assumes you are familiar with ECO handles and OclVariable components.

It is also assumed that you are designing a form that will allow the user to search a collection of `Person` objects, given all or part of a last name. The `Person` class contains an attribute called `lastName`. These have already been designed on the class diagram.

The form already contains:

- A text box component named `searchString`. The user will enter the last name search string into this text box.
- A grid component named `gPersonDisplayGrid`. The grid will display all persons whose last name matches the text entered into the `searchString` text box.
- An ECO ExpressionHandle named `ehAllPersons`. This handle will be used as the datasource for the `gPersonDisplayGrid`.

To configure the VariableHandle component

1. Drop a VariableHandle component onto the form.
2. Set the Name property of the VariableHandle to `vhLastName`.
3. Set the EcoSpaceType property of the VariableHandle to the ECO space type for your application.
4. Set the StaticValueTypeName property of the VariableHandle to `System.String`.
5. In code, set the value of the EcoSpace property of the `vhLastName` component. You can perform this step in the EcoSpace property accessor for the ECO enabled windows form.

   Add the line of code to the property accessor as shown:

```
[Delphi]

function TWinForm.get_EcoSpace: TBldOwnEcoSpace;
begin
  if not Assigned(fEcoSpace) then
  begin
    fEcoSpace := TBldOwnEcoSpace.Create;
    rhRoot.EcoSpace := fEcoSpace;

    // Set the VariableHandle's EcoSpace property
    vhLastName.EcoSpace := fEcoSpace;
  end;
  result := fEcoSpace;
end;
```

```
[C#]

public Project11EcoSpace EcoSpace {
            get {
                if (ecoSpace == null)
                {
                    ecoSpace = new Project11EcoSpace();
                    rhRoot.EcoSpace = ecoSpace;

     // Set the VariableHandle's EcoSpace property
     vhLastName.EcoSpace = ecoSpace;
                }
```

```
                return ecoSpace;
            }
        }
```

The vhLastNameVariableHandle will get its value from the searchString text box.

To retrieve the search string from the text box

1. Select the searchString text box component.
2. Create an event handler for the TextChanged event.
3. In the event handler, retrieve the value with the following line of code:

```
[Delphi]
vhLastName.Element.AsObject := searchString.Text;
```

```
[C#]
vhLastName.Element.AsObject = searchString.Text;
```

You can also use the vhLastNameVariableHandle as the databinding property of the text box. If you use this method to connect the components, you do not need an event handler. You do need to make sure the VariableHandle has been assigned a value. Do this in the constructor for the form, usually by setting the initial value to an empty string.

To configure the OclVariables component

1. Place an OclVariables component onto the form.
2. Set the Name property of the OclVariables component to oclSearchVariables.
3. Click **[...]** to open the property editor for the OclVariableCollection property.
   The .NET collection editor appears.
4. Click **Add**.
5. Set the ElementHandle property of the variable to the vhLastNameVariableHandle created above.
6. Set the VariableName property to vSearchString.
7. Click **OK**.

To configure the ExpressionHandle to use the OclVariables component

1. Select the ehAllPersonsExpressionHandle component.
2. Set the Variables property of the ehAllPersonsExpressionHandle to the oclSearchVariables component created above.
3. Click the Expression property editor.
   The **OCL Expression Editor** appears.
4. In the editor, build the OCL expression

```
Person.allInstances->select(lastName.regExpMatch(vLastname))
```

5 Click **OK**.

When the OCL evaluator encounters the expression in the `ehAllPersons`ExpressionHandle, it will look up the variable named `vLastName` in the OclVariables component, and retrieve its value from the `vhLastName`VariableHandle component. The `vhLastName` component is connected to the `searchString` text box.

The `gPersonDisplayGrid` will now display a list of all person objects whose `lastName` attribute matches the text entered by the user.

Configuring the Persistence Method of an ECO Space

You may choose to persist objects in either a relational database or in an XML file. This procedure shows how to configure the ECO space to use the persistence method of your choice.

To configure the ECO Space for the chosen persistence method

1 Locate the **Enterprise Core Objects** category in the **Tool Palette**.

There are three persistence methods to choose from:

- PersistenceMapperBdp uses the Borland Data Provider classes for database connectivity.
- PersistenceMapperSqlServer uses the native .NET database connectivity classes, which are optimized for use with Microsoft SQL Server.
- PersistenceMapperXML persists objects to an XML file instead of a relational database.

Tip: It is often useful to store your objects in an XML file during initial development and prototyping, and then switch to a relational database as your model becomes more stable.

2 Place the appropriate persistence mapper onto the **ECO Space Designer** surface.

3 Click an empty part of the **ECO Space Designer** surface.

The **Object Inspector** shows the properties of the ECO Space.

4 Set the PersistenceMapper property to the persistence mapper you created above.

Note: When using the PersistenceMapperXML component, it is not necessary to create or evolve a database schema. When persisting to an XML file, all that is required is to select the component on the **ECO Space designer**, and set its FileName property.

Converting an ECO framework Project to Developer Studio 2006

You must update an ECO framework project to the format used in Developer Studio 2006. Updating involves three separate tasks:

1. Add references to new assemblies. The `PersistenceMapperXml`, `PersistenceMapperBdp`, and `PersistenceMapperSql` components have been moved to new assemblies. You must add a reference to the appropriate assembly to your project.
2. Set the `EcoCompatibilityMode` property on the `PersistenceMapper` components on the **ECO space designer**, or on the **PersistenceMapperProvider designer**. The default names of some ECO-generated tables and columns were changed in Developer Studio 2006. You can set the `EcoCompatibilityMode` property to force the framework to use the old names. This step is not required if you are using the `DefaultORMappingBuilder` component with the `PersistenceMapper`.
3. The `SyncActive` property has been moved into a new component. If you are using the `PersistenceMapperBdp` or `PersistenceMapperSqlServer` component, and you had the `SyncActive` property set to `true`, you must add a new `SyncHandler` component. Additionally, the `HistoryLength` property has also been moved to the `SyncHandler` component.
4. Update the model. Previously, the primary storage format for the model was the source code of your business classes. In Developer Studio 2006, the model is stored in a file with the name *PackageName* `.ecopkg`, where *PackageName* is the name of the ECO package that contains your business classes.

Warning: You must perform update steps one and two prior to updating the model.

Note: The CreateRetrieveCondition method of the IOclService interface has been moved to the IOclPSService.

To add references to new assemblies

1. Open the project.
2. Close all forms and designers without saving any changes.
3. In the **Project Manager**, right-click on the project node and choose **Add Reference**.
4. Add the new assemblies to the project.

   The following table shows the new assemblies. Add the appropriate assembly, depending on the components you are using.

Component	Assembly
`PersistenceMapperXml`	Borland.Eco.Persistence.Xml.dll
`PersistenceMapperBdp`	Borland.Eco.Persistence.Bdp.dll
`PersistenceMapperSqlServer`	Borland.Eco.Persistence.SqlServer.dll

If the ECO packages do not appear in the **Model View**, close the project and reopen it.

To set the EcoCompatibility property

1. Open the project.
2. In the **Project Manager**, double-click the ECO space file to open it.
3. Click the **design tab** of the ECO space.
4. On the designer, click the `PersistenceMapper` component to select it.

5 In the **Object Inspector**, set the `EcoCompatibilityMode` property to `EcoI` or `EcoII`.

To add a new SyncHandler component

1 In the **Project Manager**, double-click the `PersistenceMapperProvider` implementation source file to open it.

2 In the **Tool Palette**, click and drag a `SyncHandler` component to the **PersistenceMapperProvider designer**.

3 Set the `HistoryLength` property of the `SyncHandler` component as necessary.

4 On the **PersistenceMapperProvider designer**, click the `PersistenceMapperBdp` or `PersistenceMapperSqlServer` component.

5 In the **Object Inspector**, set the `SyncHandler` property to the `SyncHandler` component previously created on the designer.

Note: The SyncHandler component is not available in all product SKUs.

To upgrade the model

1 Open the project.

2 Choose **Project** ▸ **Build** to build the project.

3 In the **Project Manager**, double-click the ECO space file to open it.

4 Click the design tab of the ECO space.

5 Click the **ECO space designer tool**, “Convert a compiled model to ECO III ModelLayer.”

Note: Hover the mouse cursor over the **ECO space designer tool** buttons to view the tooltip caption.

The **Convert Compiled Model** dialog will appear.

6 Click **Yes** in the **Convert Compiled Model** dialog to continue, or **No** to cancel the conversion.

The **Update Code** dialog will appear.

7 Click **Yes** to update ECO-generated source code in the model, or **No** to convert the model leaving the code as it is.

Note: The code generator creates more robust code in Developer Studio 2006, so this step is highly recommended.

After conversion, the project will contain a *ProjectName* `.ecopkg` file for each ECO package in your model. The conversion tool will also add each new *ProjectName* `.ecopkg` file to the project.

Creating a New ECO Space Subclass

You can add a new ECO space subclass to a project using the ECO Space code template.

Note: The ECO wizards are available in both C# and Delphi for .NET projects. The functionality of the wizards is identical. The only difference is in the language used to generate source code files.

To create a new subclass of the EcoSpace class

1. Open an ECO project.
2. Choose **File New Other**.

   The **New Items** dialog box appears.
3. Select the **New ECO Files** category.
4. Double-click the **ECO Space** icon.

The **ECO Space** wizard will generate a source code file that contains a new subclass of the DefaultEcoSpace class.

Creating a New ECO Windows Forms Application

You can create a new ECO Windows Forms application with the **ECO Winforms Application** code template.

Note: The ECO wizards are available in both C# and Delphi for .NET projects. The functionality of the wizards is identical. The only difference is in the language used to generate source code files.

To create a new ECO Windows Forms application

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box appears.
2. Select either **Delphi for .NET Projects**, or **C# Projects**.
3. Double-click **ECO Windows Forms Application**.

   The **New Application** dialog box appears.
4. Type the name of your project, and use the **[...]** button to locate the folder where you want to place the project files.

The **ECO WinForms Application** wizard generates a new project containing the following files.

File	Description
Package_NUnit.pas	Contains the source code for the UML packages, interfaces, classes and their associations, and all other types in your model. `N` is an increasing integer.
<ProjectName>EcoSpace.pas	Contains source code for the subclass of `Borland.Eco.Handles.EcoSpace`. `<ProjectName>` is replaced with the name of your project.
WinForm.pas	Contains source code for the ECO-enabled main WinForm of the application. The main form for an ECO application provides a property that holds an instance of the ECO Space of the application. The form also contains code to automatically allocate an instance the ECO Space class.
Borland.Eco.Windows.Forms.dll Borland.Eco.Handles.dll Borland.Eco.Interfaces.dll Borland.Eco.Ocl.ParserCore.dll Borland.Eco.Persistence.dll	The **ECO WinForms Application** wizard automatically adds references to these assemblies, and they must be distributed with your application along with all other referenced assemblies.

The generated ECO enabled form also contains the following extender providers for buttons, listboxes, and grids.

Extender Provider	Purpose
`EcoGlobalActions`	Extends buttons with the EcoAction property. This property allows buttons to perform database-oriented operations (such as Update, Undo, and Redo) without writing code.
`EcoAutoForms`	Extends grids and list boxes with the EcoAutoForm property. If the EcoAutoForm property is set to True, double-clicking the grid or list box will open an automatically-generated form that describes the selected object.
`EcoListActions`	Extends buttons with the CurrencyManager and EcoListAction properties. The EcoListAction property allows buttons to perform list-oriented operations (such as Add, Delete, MoveFirst, and MoveNext) without writing code.

	The CurrencyManager property must be set to the CurrencyManager used by the ExpressionHandle that holds the list on which to operate.
`EcoDragDrop`	Extends grids and list boxes with the EcoDragSource and EcoDropTarget properties. If EcoDragSource is set to true, you can drag objects from the listbox or grid. If EcoDropTarget is set to True, you can drop an object onto the control. The object must conform to the target list. For example, you can drag a `Person` object into a `Customer` list, but you cannot drag a `Person` object into a `City` list.

Creating a PersistenceMapperProvider

You can add a PersistenceMapperProvider to your project using the **ECO PersistenceMapperProvider** code template. A PersistenceMapperProvider binds the persistence components and their configuration, allowing them to be shared by more than one ECO space. The PersistenceMapperProvider is thread-safe and remotable, so multiple instances of an ECO space can connect to a single provider.

This procedure assumes you have an existing ECO application project open. It can be either an ECO WinForms application, or an ECO ASP.NET Web application. Please refer to the links below for more information on creating ECO application projects.

To create a new PersistenceMapperProvider

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box appears.

2. Select the **New ECO Files** category.
3. Double-click **ECO Persistence Mapper Provider**.

The **Persistence Mapper Provider** wizard creates a source file called EcoPersistenceMapperProvider.pas (or EcoPersistenceMapperProvider.cs, depending on the type of project you have open).

Creating an Association Class on an ECO Class Diagram

Associations can have classes associated with them. An association class is useful when the association itself has attributes.

This procedure assumes you have an existing **ECO class diagram** with two classes and an association between them.

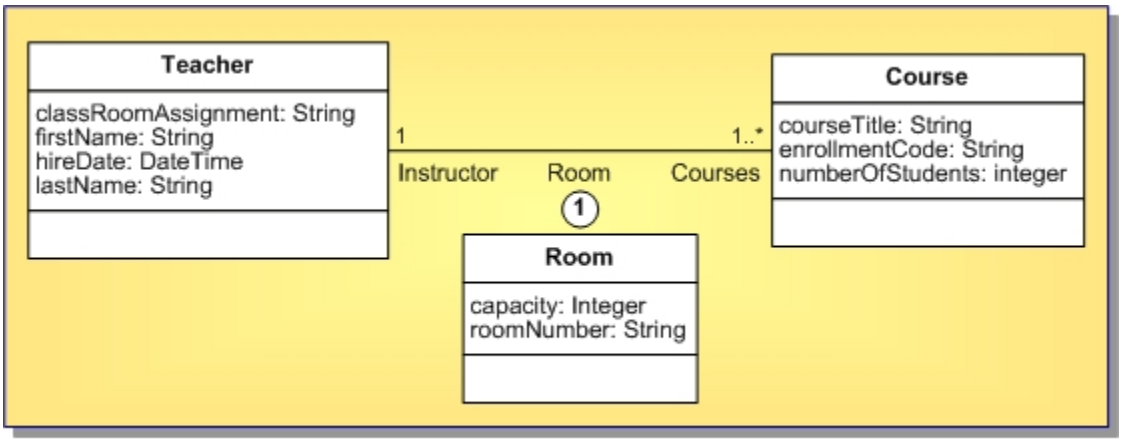
To create an association class

1. Add a new ECO class to an **ECO class diagram**.

   This class will be the association class.
2. Select the association for which you want to create an association class.
3. In the **Object Inspector**, click the property editor ellipses for the `Association Class` property.

   The **Choose Class to Instantiate** dialog box appears.
4. Expand the tree node of the ECO package that contains the association class you just added.
5. Select the association class in the tree.
6. Click OK.

The following diagram shows an **ECO class diagram** with two classes called `Teacher` and `Course`, and an association class called `Room`.



(1) Association with association class, Room

Creating an ECO ASP.NET Application

This procedure describes how to create a basic ECO ASP.NET application. Please refer to the topics below for more information on ASP.NET and the ECO framework.

To create an ECO ASP.NET application

1. Choose **File** ▸ **New** ▸ **Other**.
   The **New Items** dialog box appears..
2. Select **ECO ASP.NET Application** for either Delphi for .NET or C#.
3. In the **Name** field, enter the name of your project.
4. In the **Location** field, accept the default path or enter another project path.

   Tip: Most ASP.NET projects reside in the IIS directory Inetpub\wwwroot.

To change Web server settings (optional)

1. In the **New ECO ASP.NET Application** dialog box, click **View Server Options**
   The dialog box expands to show additional server options.
2. Set the read and write attributes of the project as needed or accept the defaults.

   Tip: In most cases, the default settings will be sufficient.

3. Click **OK**.
   The **Web Forms Designer** appears.

Creating an ECO framework State Machine Diagram

State machine diagrams are associated with classes in your model. Each class may have one or more state diagrams associated with it.

To create a new ECO state machine diagram

1. Choose **View** ▸ **Model View**.

   The **Model View** displays.

2. Expand the ECO package that contains the class for which you want to add a state diagram.

   Note: You must expand the ECO package in the **Model View**. An ECO state machine diagram must be added to the class in the ECO package, not from the C# namespace or Delphi unit.

3. Right click on the class and choose **Add** ▸ **ECO State Machine** from the menu.

A new ECO state machine diagram will be added to the model for the selected class.

To develop a state machine diagram

1. Add states and substates to the diagram.

   Adding States and Substates to an ECO State Machine Diagram

2. Add trigger methods to your classes.

   Adding a Trigger Method to an ECO Class

3. Add guard expressions to state transitions.

   Adding a Guard Expression to a State Transition

4. Add entry and exit actions to states.

   Adding Entry and Exit Actions to a State

5. Add state transition effects.

   Adding an Effect to a State Transition

6. Add regions to a state to use concurrent substates.

   Adding a Region to a State

As you work with the diagram, select elements and set their properties using the **Object Inspector**.

Creating an ECO Package in a DLL

This topic describes how to create an ECO package in a DLL, rather than a full ECO application.

The **ECO Package in a DLL** wizard generates an ECO UML package and associated source files, but it does not create an ECO space class. You can reference the ECO packages in the DLL in an ECO application. This will make the model available in read-only mode in the application that references the packages. You may draw inheritance links and unidirectional associations to the referenced classes, but you may not add attributes or methods.

Note: The ECO wizards are available in both C# and Delphi for .NET projects. The functionality of the wizards is identical. The only difference is in the language used to generate source code files.

To create an ECO Package in a DLL

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box appears.
2. Select either the **Delphi for .NET Projects** category, or the **C# Projects** category.
3. Double-click **ECO Package in a DLL** (C#), or **ECO Package in package** (Delphi for .NET).

A new project is created with one default ECO package.

Creating an Empty InterBase Database

To create an empty InterBase database using the IBConsole program

1. Start the InterBase Console program, IBConsole.
2. Logon to the server where you want the new database to be created.

   Note: Under the Databases node you will see a list of databases that reside on that server.

3. Right-click the Databases node and choose **Create Database**.
4. Type the path and file name of the database.
5. Customize any database parameters you wish to change.
6. Click **OK**.

   The new database will be displayed under the Databases node in the IBConsole window.

Creating an Event Derived Column

This procedure assumes you are familiar with ECO handles and columns.

If the value of a column cannot be computed using OCL, you can derive the value in source code by creating an event derived column.

To create an event derived column

1. Add a column to an existing handle.

   Please refer to the procedure *Adding Columns and Nestings to an ECO Handle* for more information.
2. In the **Column Collection editor**, set the EventDerivedValue property of the column to True.
3. Click OK.
4. Select the handle on the form designer.
5. Select the **Events** tab in the **Object Inspector**.
6. Add an event handler for the DeriveValue event.

There is one event handler for all the event derived columns in the column collection of the handle. In the event handler code, you can examine the Name property of the DeriveEventArgs parameter to determine which column value is being requested.

Pass the computed value of the column back in the ResultElement property of the DeriveEventArgs parameter.

Deploying an ECO framework Application

To deploy an ECO application

1. Open the project.
2. Choose **View** ▸ **Project Manager**.
3. Select the appropriate compiler settings in the **Project Options** dialog box.

   Note: You must set the appropriate build settings on each project in the project group of your application.

4. Select **Project** ▸ **Build <Project Name>** where *<Project Name>* is the actual name of your project to build your application.

   The build targets for each project in the project group will be generated per their own respective project settings.

   Referenced assemblies that have their Copy Local setting checked will be copied to the output directory of the project that references them.

In addition to the other assemblies your project references, there are ECO-specific assemblies that must be deployed with all ECO applications. The tables below show the ECO assemblies that are required, depending on the deployment scenario.

ECO assemblies to be deployed in all cases

Borland.Eco.Core.dll
Borland.Eco.Handles.dll
Borland.Eco.Interfaces.dll
Borland.Eco.Ocl.ParserCore.dll
Borland.Delphi.dll Note: this assembly is required even for C# projects.

Additional deployment scenarios

Scenario	Requirement
Projects that use persistence to a relational database	Borland.Eco.Persistence.dll, and Borland.Eco.Persistence.Bdp.dll, or Borland.Eco.Persistence.SqlServer.dll
Projects that use multiple databases and ECO space synchronization	Borland.Eco.Persistence.Multi.dll
Projects that use persistence to XML file only	Borland.Eco.Persistence.Xml.dll The Borland.Eco.Persistence.dll assembly is not required in this case.
Windows Forms projects	Borland.Eco.Windows.Forms
ECO ASP.NET projects	Borland.Eco.Web.dll
Projects that use BDP	Borland.Data.Common, Borland.Data.Provider You must also deploy the assemblies required for your particular database, such as Borland.Data.Interbase.dll and bdpint20.dll, for InterBase.

Projects that use ECO DBWebControls	Borland.Data.Web.dll, Borland.Data.Web.Eco.dll

The Developer Studio 2006 installer deploys these assemblies into the .NET Global Assembly Cache (GAC). The GAC cannot be viewed or manipulated directly. Copies of these files are kept with other shared assemblies in the Developer Studio 2006 Common Files folder. The default path to this location is \Program Files\Common Files \Borland Shared\BDS\Shared Assemblies\<version>, where <version> is the version number of Developer Studio 2006 that is installed on the development machine.

You can deploy the ECO assemblies into the GAC, or you can deploy them into the installation directory of the application. If you will be deploying multiple ECO applications, it is best to deploy them as shared assemblies.

Deriving an Attribute in Source Code

You can derive the value of an attribute in source code when you cannot derive it in OCL. To do this, you must implement a specific design pattern so that the method that computes the attribute value can be called by the framework.

To create the source code method

1. In the **ECO class diagram**, select the attribute in the class that you want to derive.
2. In the **Object Inspector**, set the derived property of the attribute to true.
3. Make sure the DerivationOCL property of the element is left blank.
4. Create a method in your class with the following signature:

```
[Delphi]

function attributeNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
```

```
[C#]

System.Object attributeNameDeriveAndSubscribe(ISubscriber reevaluateSubscriber,
ISubscriber resubscribeSubscriber);
```

5. Replace `attributeName` with the name of the attribute whose value you are computing. For example if you are computing the value of an attribute called `fullName`, the method signature would be:

```
[Delphi]

function fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
```

```
[C#]

System.Object fullNameDeriveAndSubscribe(ISubscriber reevaluateSubscriber, ISubscriber
resubscribeSubscriber);
```

6. Within the implementation of the `DeriveAndSubscribe` method, perform the calculations necessary to compute the value of the attribute. To compute the value of the full name of a person, you might write code such as the following:

```
[Delphi]

function Person.fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
Var
   fullName : String;
begin
       fullName := firstName + ' ' + lastName;
       result := fullName;
end;
```

```
[C#]

System.Object fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber)
{
    string   fullName;

        fullName = firstName + " " + lastName;
        return fullName;
}
```

Now you must determine which elements need subscriptions. In the computation of the `fullName` attribute, you used the `firstName` and `lastName` attributes of the `Person` class. Therefore, you must place subscriptions on these two attributes so that the value of the `fullName` attribute will be reevaluated when a change occurs to either a person's first or last name.

To place a subscription, call the method SubscribeToValue. This method is implemented by the framework for all IElement types, so you must cast the object using the AsIObject method, and then you can place the subscription.

To place a subscription on an attribute, call the SubscribeToValue method in the `DeriveAndSubscribe` method, as shown below (SubscribeToValue is a method of the IElement interface).

Notice that in this example, you only need a reevaluate subscription, since you are interested only in the atomic values, `firstName` and `lastName`.

```
[Delphi]

function Person.fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber) : System.Object;
Var
   fullName : String;
begin
        fullName := firstName + lastName;

       // Subscribe to the firstName and lastName attributes.
       AsIObject.Properties['firstName'].SubscribeToValue(reevaluateSubscriber);
       AsIObject.Properties['lastName'].SubscribeToValue(reevaluateSubscriber);

       result := fullName;
end;
```

```
[C#]

System.Object fullNameDeriveAndSubscribe(reevaluateSubscriber : ISubscriber;
resubscribeSubscriber : ISubscriber)
{
    string   fullName;

        fullName := firstName + lastName;

        // Subscribe to the firstName and lastName attributes.
        AsIObject.Properties['firstName'].SubscribeToValue(reevaluateSubscriber);
        AsIObject.Properties['lastName'].SubscribeToValue(reevaluateSubscriber);

        return fullName;
}
```

Tip: The Properties of an IObject can be indexed using the name of the attribute you are interested in.

Generating a Model and OR Mapping from an Existing Database

The **Wrap existing database** tool on the **ECO Space Designer**, produces the following:

- An ECO package containing the model with classes, attributes, and associations
- Source files for generated classes
- An XML file describing the generated OR mapping

To wrap an existing database

1 Configure the persistence components of the ECO space to use the existing database.

Using the ECO Space Designer

2 On the **ECO space designer**, click the **Wrap Existing Database with ECO** tool.

3 In step one of the wizard, select an existing XML file that contains an OR mapping for the database.

This step is optional. If you do not have an existing XML file, click **Next**.

> **Note:** If you specify an XML file, the wizard will attempt to merge the contents of the XML file with the database schema, to create a single OR mapping file.

4 In step two of the wizard, use the **Database table tree** to include or exclude tables or table columns from the generated mapping.

> **Note:** You may further customize the mapping by selecting a node in the tree and setting its properties in the **Node properties** list.

5 In step three of the wizard, use the **Classes and properties tree** to include or exclude classes or class properties from the generated mapping.

> **Note:** You may further customize the mapping by selecting a node in the tree and setting its properties in the **Node properties** list.
>
> The last step of the wizard displays a report of what will take place when you click **Finish**.

6 Place a FileMappingProvider component onto the **ECO Space Designer**.

7 Enter the name of the XML file produced above in the FileName property of the FileMappingProvider component.

> **Note:** The XML mapping file must be distributed with the application. The path to this file is relative to the executable file.

8 Set the RunTimeMappingProvider property of the persistence mapper component to the FileMappingProvider created above.

9 In the **ECO space designer toolbar**, click **Select Packages** tool.

The **Select Packages** dialog displays.

Use the **Select Packages** dialog to add ECO packages to the ECO space.

The new ECO UML packages generated in the source file are displayed in the **Model View** window. You can open the class diagram and make changes as you would with any ECO package.

You must make modifications to fine tune the generated model and XML mapping if they do not accurately match the database schema. These changes must be made on the class diagram surface and in the XML mapping file.

Note: If you need to reverse engineer multiple databases, repeat this procedure once for each database. You will need to add and configure separate persistence mappers and file mapping provider components for each database.

Implementing a Subclass of SubscriberAdapterBase

This procedure assumes you are familiar with the basic concepts of working with ECO subscriptions.

Rather than implementing the `ISubscriber` interface directly, you should create a small, private utility subclass of SubscriberAdapterBase. SubscriberAdapterBase is an abstract class that handles most of the implementation details of the `ISubscriber` interface.

In your subclass, you only have to implement the DoReceive method to respond to the subscription event.

To implement a subclass of SubscriberAdapterBase

1 Create a utility class within the class that will respond to the subscription event, which might look like the following:

```
using System;
using Borland.Eco.Subscription;

private class MySubscribingClass {

         private class MySubscriberAdapter : SubscriberAdapterBase {

           // Notice the actual subscriber class (MySubscribingClass) is
           // passed on to the SubscriberAdapterBase class.
           public MySubscriberAdapter(object subscriber) : base(subscriber)
           { }

  }

}
```

2 Implement the DoReceive method in the utility class.

```
private class MySubscriberAdapter : SubscriberAdapterBase {

     public MySubscriberAdapter(object subscriber) : base(subscriber)
     { }

     protected override void DoReceive(object  sender,EventArgs e) {


       // ActualSubscriber is a property of SubscriberAdapterBase.
       (ActualSubscriber as MySubscribingClass).RespondToEvent();
     }
}
```

In the previous code, the method `ResondToEvent` is implemented in the outer class, `MySubscriberClass`.

3 Implement the `RespondToEvent` method in the outer class.

```
private class MySubscribingClass {

         private class MySubscriberAdapter : SubscriberAdapterBase {

               protected override void DoReceive(object sender, EventArgs e) {
```

```
                  // ActualSubscriber is a property of SubscriberAdapterBase.
                (ActualSubscriber as MySubscribingClass).RespondToEvent();
                }

                public MySubscriberAdapter(object subscriber) : base(subscriber)
                { }

        }


        private void RespondToEvent() {

            // Add code to handle the event

        }
}
```

4 Write code to place a subscription.

In the following code, a field is added to hold an instance of the subscriber adapter, and a new method is defined to place the subscription. This example shows how to use the IExtentService interface to receive subscription events whenever objects of a given class are created.

```
private class MySubscribingClass {

        private class MySubscriberAdapter : SubscriberAdapterBase {

              protected override void DoReceive(object sender, EventArgs e) {

                 // ActualSubscriber is a property of SubscriberAdapterBase.
                (ActualSubscriber as MySubscribingClass).RespondToEvent();
                }

                public MySubscriberAdapter(object subscriber) : base(subscriber)
                { }

        }

        private void RespondToEvent() {

            // Add code to handle the event

        }

        // Add a field to hold the subscriber adapter.
        private MySubscriberAdapter myAdapter = null;

        public void SubscribeToObject(IExtentService extentService, IClass
subscribeToClass) {

            // Drop old subscriptions if any (Deactivate is a method of
SubsciberAdapterBase)
            if(myAdapter != null)
               myAdapter.Deactivate();

            // Create an instance of the private subscriber adapter
            myAdapter = new MySubscriberAdapter(this);

            // Place a subscription
```

```
            extentService.SubscribeToObjectAdded(myAdapter, subscribeToClass);
        }
}
```

The DoReceive method of `MySubscriberAdapter` will be called whenever new objects of the type passed to `SubscribeToObject` are created.

Regenerating and Updating ECO Source Code

If you have turned off automatic code regeneration in the **Tools** ▸ **Options** dialog box, then you need to regenerate or update your ECO source code manually. You may either completely recreate all ECO source code, or update only those classes that you have changed since the last update.

Source code regeneration changes the following ECO-generated code elements

- Implementation of list interfaces
- The `Initialize`, `Deinitialize`, `get_MemberByIndex`, and `set_MemberByIndex` methods.
- ECO properties
- Custom attributes, for example, `[UmlElement()]`
- Methods with the `Is trigger` or `Is query` attribute set to `true`.

Note: Code regeneration does not change those elements that are expected to have user-written code. This includes attributes that have the `HasUserCode` attribute set to `true`, methods not marked as state machine triggers or queries, public constructors, and methods that have a body specified in ECO Action Language.

To completely regenerate ECO source code

1. Choose **View** ▸ **Model View** to open the **Model View** for your project.
2. Right-click on the top-level node in the **Model View tree** and choose **Regenerate ECO source code**.

All source code files for ECO classes and ECO packages will be regenerated except for those parts noted above.

Tip: You may also regenerate ECO source code by clicking the **Regenerate ECO source code** toolbar button in the **Model View**.

To update ECO source code

1. Choose **View** ▸ **Model View** to open the **Model View**.
2. Right-click on the top-level node in the **Model View tree** and choose **Update ECO source code**.

The update option makes as few changes as possible to ECO source code. Like the regenerate option, the update option only changes source code as noted above.

Tip: You may also update ECO source code by clicking the **Update ECO source code** toolbar button in the **Model View**.

Selecting ECO UML Packages

This procedure describes how to select the ECO UML packages that you want to use in the ECO space. Classes that reside in packages other than those selected cannot be stored in the ECO space.

To select UML packages

1. Click **Select Packages** on the **ECO space designer**. The **Select packages** dialog box appears.

   A full list of available UML packages is shown in the **Available Packages** list box. UML packages that are already managed in the ECO Space are shown in the **Selected Packages** list box.

2. To add a single UML package, select it in the list, and click **[<]**.

   To add all available packages, click **[<<]**.

3. To remove a single package from the ECO Space, select the package in the **Selected Packages** list, and click **[>]**.

   To remove all selected packages from the ECO space, click **[>>]**.

Using a Custom Object-Relational Mapping File

You can use a custom OR mapping file with the built-in database schema evolution tools available on the **ECO Space Designer**.

This procedure assumes you are familiar with using the **ECO Space Designer**.

To specify a custom OR mapping file

1. Place a FileMappingProvider component onto the **ECO Space designer**.
2. Set the FileName property of the FileMappingProvider component to the name of the custom OR mapping file.
3. Select the persistence mapper component of the ECO space.
4. Set the RuntimeMappingProvider property of the persistence mapper to the FileMappingProvider component.

After you make changes to the custom OR mapping file, there are two ways to evolve the database schema. Changes to the schema can be made manually, using the appropriate database tools supplied by your vendor, or you can use the database evolution tool on the **ECO Space Designer**. If you use the **ECO Space Designer**, you must set the old and new OR mapping provider properties of the persistence mapper component, as described below.

To use ECO database evolution with a custom OR mapping file

1. Save a copy of your current OR mapping file before making any changes.

   This file is now the old mapping file.
2. Place a second FileMappingProvider component onto the **ECO Space Designer**.
3. Set the FileName property of the FileMappingProvider to the copy of the OR mapping file you created in step 1..
4. Select the persistence mapper component on the **ECO space designer**.
5. Set the OldMappingProvider property of the persistence mapper to the FileMappingProvider you created in step 2.
6. Set the NewMappingProvider property of the persistence mapper to the same FileMappingProvider as the RuntimeMappingProvider property.

Tip: If you make changes to the database schema manually, you only need to set the RuntimeMappingProvider property of the persistence mapper.

Using the ECO Space Designer

An ECO Space is a container for the runtime instances of the classes in your model. The **ECO Space Designer** lets you select UML packages from your model, choose the persistence mechanism for objects, create or evolve the database schema, and perform designtime validation of the model.

You cannot work directly with the class EcoSpace. Instead, the IDE automatically creates a subclass of the EcoSpace class for you when you create a new ECO application. If you have imported a model from another tool, such as Bold for Delphi or Together Control Center, you can add an ECO Space to your project using the **ECO Space** wizard in the **New Items** dialog box.

The ECO Space class for your application is implemented in one source file. The default source file name is EcoSpace.cs or EcoSpace.pas. To open the **ECO Space Designer**, double-click the source code file in the **Project Manager** window, and then click the **Design** tab. This document describes the basic procedure for configuring an ECO Space. Each step is then explained in more detail in the following sections.

Warning: You must compile or build your application prior to using the **ECO Space Designer**. The ECO framework makes extensive use of .NET custom attributes, and building your application ensures that the designer is working with the correct assembly metadata.

To configure an ECO Space

1. Select the UML packages containing the classes that you want to exist in the ECO space.

   Selecting ECO UML Packages

2. Choose a persistence method, either a relational database, or XML file.

   Configuring the Persistence Method of an ECO Space

3. On the **ECO space designer**, click the **Validate model** tool to perform consistency checks on the model.

   The IDE will perform a number of checks to make sure the model is well-formed. For example, OCL expressions are checked to make sure they are valid.

4. If you are using an RDBMS, create an empty database.

   Note: The exact procedure will vary depending on your database vendor. This procedure creates an InterBase database as an example.

   Creating an Empty InterBase Database

5. Add a connection handle component to the ECO Space and configure its connection string.

   Adding and Configuring a Connection Handle on an ECO Space

6. If you are using an RDBMS and you are starting from scratch, create the initial database schema by clicking the **Generate Schema** tool on the **ECO space designer**. Otherwise, if you have made changes to the model, or you want to add or remove a UML package, use the **Evolve Schema** tool to update the existing database schema.

Note: During creation or evolution of the database schema, the **ECO** tab in the **Message** pane will display status messages and results of the operation.

Warning: When a class or attribute is deleted and the database schema evolved, the corresponding columns are removed from the database and the data is lost. You will have a chance to cancel the operation before proceeding.

Using the Expression Editor to Build OCL and ECO Action Language Expressions

There are two expression editors available in Developer Studio 2006: The Object Constraint Language (OCL) Expression Editor, and the ECO Action Language Expression Editor. Some ECO components and model elements have properties whose values are expressed in OCL or in ECO Action Language. The **Expression Editor** is a property editor for those components and model elements. It is invoked from the **Object Inspector** by clicking the **[...]** button for the property.

The two expression editors are essentially the same, except the **ECO Action Language Expression Editor** features extensions that allow you to change values in the model. In contrast, OCL expressions are used only to calculate values or query the ECO space for objects.

There are two ways to build an expression: Type directly into the left-hand pane of the editor, or, use the right-hand pane to build the expression from model types and OCL or ECO Action Language operations. The right-hand pane shows the types, attributes, and operations that are valid in the current context. As you build the expression, the editor checks syntax and reports errors in the **Parser Message** tab.

Note: When editing the Expression property of an ExpressionHandle component, if the model types and operations do not appear in the right-hand pane, make sure the EcoSpaceType property for the root handle is set to a valid ECO Space. In addition, check the **Parser Message** tab for any errors encountered in the model itself.

To open the Expression Editor

1. Select the component on the **Windows Form designer**, or the model element on the diagram.
2. In the **Object Inspector**, click **[...]** for the property.

   The **Expression Editor** opens.

   Depending on the property, the **Expression Editor** will build an OCL expression, or an ECO Action Language expression.

The following table shows the actions available in the editor:

Desired action	Editor interface
To add a type or operation to the current expression	Double-click the type or expression in the right-hand pane. The type or operation will be added to the current expression in the left-hand pane.
To remove the last type or operation added to the expression	Click the **Remove** button.
To clear the current expression	Click the **Clear** button.
To toggle the view in the right-hand pane between list and tree modes	Click the **Switch between list and tree view** button, located above the right-hand pane.
To view class inheritance in tree-view mode	Click the **Show classes hierarchical** button, located above the right-hand pane.
To hide or show return types of operations in the right-hand pane	Click the **Show return types** button, located above the right-hand pane.
To hide or show the class in which attributes are defined	Click the **Show defining class** button, located above the right-hand pane.

To use the OCL Expression Editor when adding columns

1. With an expression handle selected, click **[...]** for the **Columns** property in the **Object Inspector**.

The **Column Collection Editor** opens.

2. Click **Add OclColumn**.
3. In the **Column Collection Editor**, click **[...]** for the **Expression** field.

   The **OCL Expression Editor** opens.
4. Construct your expression by double-clicking elements from the right-hand pane of the **OCL Expression Editor** until the expression is complete.
5. Click **OK** to close the **OCL Expression Editor**.
6. Click **OK** to close the **Column Collection Editor**.

   At runtime, a new column is added to any data grid component that is linked to the expression handle.

Using the PersistenceMapperProvider Designer

This procedure assumes you have an existing ECO ASP.NET or ECO Windows Forms project open.

A `PersistenceMapperProvider` component allows you to share the same PersistenceMapper among several ECO Space instances. The **PersistenceMapperProvider designer** is very similar to the **ECO space designer**, however, the **PersistenceMapperProvider designer** only allows for creation and evolution of database schema.

To configure the PersistenceMapperProvider

1. Add a `PersistenceMapperProvider` component to your project.

   After adding the component, select its **Design** tab in the editor pane.

   Creating a PersistenceMapperProvider
2. Choose a persistence method, either a relational database, or XML file.

   Configuring the Persistence Method of an ECO Space
3. Click on an empty part of the **PersistenceMapperProvider Designer** surface.

   The **Object Inspector** shows the properties of the `PersistenceMapperProvider`.
4. Set the PersistenceMapper property to the persistence mapper you created previously.
5. Compile the project.
6. Select the ECO space from the dropdown list on the EcoSpaceType property of the `PersistenceMapperProvider`.

   Note: You must compile the application before the ECO space will appear in the dropdown list.
7. If you are using an RDBMS, create an empty database.

   Note: The exact procedure will vary depending on your database vendor. This procedure creates an InterBase database as an example.

   Creating an Empty InterBase Database
8. Add a connection handle component to the ECO Space and configure its connection string.

   Adding and Configuring a Connection Handle on an ECO Space
9. If you are using an RDBMS and you are starting from scratch, create the initial database schema by clicking the **Generate Schema** tool on the **ECO space designer**. Otherwise, if you have made changes to the model, or you want to add or remove a UML package, use the **Evolve Schema** tool to update the existing database schema.
10. If you are creating a server that will share its ECO space with multiple clients, add a PersistenceMapperSharer component to the ECO space.

    Adding a PersistenceMapperSharer to an ECO Space
11. If you are creating a client application that will connect to an ECO space provided by a server, add a PersistenceMapperClient component to the ECO space.

    Adding a PersistenceMapperClient to an ECO Space

Interoperable Applications Procedures

Adding a Reference to a COM Server

To Add a Reference to a COM Server

1. In the **Project Manager**, right-click the **References** tree node of your project, and select **Add Reference**.
2. In the **Add Reference** dialog box, click the **COM Imports** tab.

   The IDE will scan the system registry for all registered type libraries and COM servers.
3. Select the item or items you want to reference in your project.

   Tip: You can individually select multiple items from the list by holding down the CTRL key as you click each item. To select a range of items, select the first item, then hold down the SHIFT key as you select the second item.
4. Click the **Add Reference** button.

   All of the items you selected will appear in the **New References** list in the bottom portion of the dialog.

   Tip: You can remove items from the **New References** list. Select the item or items and click the **Remove** button.
5. If the COM component you want to reference does not appear in the list, click the **Browse** button to add an explicit reference to it.
6. In the **Select a reference** dialog box, navigate to the folder where the component is located.
7. Select it, and click **Open**.
8. When you have selected all of the COM servers you wish to add, click **OK**.

After you click the **OK** button in the **Add Reference** dialog, the IDE will generate interop assemblies for each item you selected (unless a Primary Interop Assembly has already been created). These assemblies will be named Interop.LibraryName.dll, where LibraryName is the name of the component's type library (note this name might differ from the control's DLL file name). The generated assemblies will be stored in a folder called COMImports, under your project directory. Each generated interop assembly will be set to Copy Local, meaning, when the project is built, the assembly will be copied to the build target folder automatically.

The COMImports folder might not exist, for example, if you move the project to a new machine, or if you delete it on the machine where the project resides. If the COMImports folder does not exist when the project is reopened, the IDE will recreate it and regenerate the interop assemblies. In order for this to work, the COM servers must first be registered on the machine where the project resides.

If a Primary Interop Assembly for the COM server exists, the IDE will not generate a new interop assembly. Instead a reference to the Primary Interop Assembly will be added, and the Copy Local setting will be turned off, since Primary Interop Assemblies are deployed in the Global Assembly Cache.

Note: To see the Copy Local setting on any referenced assembly, right click the mouse on the assembly in the **Project Manager**. The Copy Local setting is an item on the context menu.

The project will still retain references to the interop assemblies, even if the COMImports folder could not be regenerated. In this case, the **Project Manager** will highlight the referenced assembly to indicate that it currently does not exist on the machine.

Adding an ActiveX Control to the Tool Palette

To Add an ActiveX Control to the Tool Palette

1. Choose **Component** ▸ **Installed .NET Components** from the menu.

   The **Installed .NET Components** dialog box appears.

2. Click the **ActiveX Components** tab.

   The IDE will scan the Windows registry for all available ActiveX controls.

3. Click the check box next to the control names you want to add to the **Tool Palette**.

   Tip: To remove a control from the **Tool Palette**, uncheck the check box next to the control's name.

4. If a control you want to add does not appear in the list, click the **Select an ActiveX Component** button to add an explicit reference to it.
5. In the **Browse to select an ActiveX Component** dialog box, navigate to the folder where the control is located.
6. Select the control, and click **Open**.
7. Click **OK**.

   The ActiveX Controls you selected will appear on the **Tool Palette**.

By default, ActiveX Controls appear in the ActiveX category. You can then drag the ActiveX Control from the **Tool Palette**, onto your form. When you do so, the IDE will generate an interop assembly and an ActiveX control wrapper assembly for each item you selected (unless a Primary Interop Assembly has already been created). These assemblies will be named Interop.LibraryName.dll, and AxInterop.LibraryName.dll where LibraryName is the name of the component's type library (note this name might differ from the control's DLL file name). The generated assemblies will be stored in a folder called COMImports, under your project directory. Each generated interop and ActiveX control wrapper assembly will be set to Copy Local, meaning, when the project is built, the assembly will be copied to the build target folder automatically.

The COMImports folder might not exist, for example, if you move the project to a new machine, or if you delete it on the machine where the project resides. If the COMImports folder does not exist when the project is reopened, the IDE will recreate it and regenerate the interop assemblies. In order for this to work, the COM servers must first be registered on the machine where the project resides.

If a Primary Interop Assembly for the ActiveX control exists, the IDE will not generate new interop assemblies. Instead a reference to the Primary Interop Assembly will be added, and the Copy Local setting will be turned off, since Primary Interop Assemblies are deployed in the Global Assembly Cache.

Note: To see the Copy Local setting on any referenced assembly, right-click the assembly in the **Project Manager**. The Copy Local setting is an item on the context menu.

The project will still retain references to the interop assemblies, even if the COMImports folder could not be regenerated. In this case, the **Project Manager** will highlight the referenced assembly to indicate that it currently does not exist on the machine.

Installing Janeva Compilers in the Tools Menu

You can install the Janeva compilers into the Tools Menu, if you want to access them in a separate command window, rather than compiling your IDL directly from within the **Code Editor**.

To install a compiler to the Tools menu

1. Choose **Tools** ▸ **Configure Tools**.

   This displays the **Tool Options** dialog.
2. Click **Add**.
3. In the **Title:** text box, enter the name you want to appear on the **Tools** menu.
4. In the **Program:** text box, enter the path and name of the *idl2cs* or the *java2cs* executables.

   Tip: You can locate the program and its path, as well as its working directory, by clicking the **Browse** button to display the **Select Transfer Item** dialog. Navigate to the program executable and double-click on it to add its name to the **Program:** text box.
5. In the **Working dir:** text box, enter the name of the working directory.
6. In the **Parameters:** text box, enter the parameters you want to pass to the program when the user selects the menu command.
7. To enter pre-defined macros in the **Parameters:** text box, click **Macros** and double-click on the macro names you want to pass to the program when the user selects the menu command.
8. Click **OK**.

   The name you specified in the **Title:** text box appears in the **Tools** menu. The user can select this command to execute the compiler in a command window.

VCL for .NET Procedures

Building a VCL Forms Application

The following procedure illustrates the essential steps to building a VCL Forms application using Developer Studio 2006.

To create a VCL Form

1. Choose **File** ▸ **New** ▸ **Other**.
   The **New Items** dialog appears.
2. Select **Delphi for .NET Projects**.
3. Double-click **VCL Forms Application**.
   The **VCL Forms Designer** displays.
4. From the **Tool Palette**, place components onto the form to create the user interface.
5. Write the code for the controls.

To associate code with a control

1. Double-click a component on the form. The **Code Editor** displays, cursor in place within the event handler block.
2. Code your application logic.
3. Save and compile the application.

Building a VCL Forms dbExpress.NET Database Application

The following procedures describe how to build a dbExpress database application.

Building a VCL Forms dbExpress.NET application consists of the following major steps:

1. Set up the database connection.
2. Set up the unidirectional dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add a dbExpress connection component

1. Choose **File** ▸ **New** ▸ **VCL Forms Application**.

   The **VCL Forms Designer** displays.
2. From the **dbExpress** category of the **Tool Palette**, place a TSQLConnection component on the form.
3. Double-click the TSQLConnection component to display the **Connection Editor**.
4. In the **Connection Editor**, set the **Connection Name** list to IBConnection.
5. In the **Connections Setting** box, specify the path to the InterBase database file called employee.gdb in the Database field.

   By default, the file is located in C:\Program Files\Common Files\Borland Shared\Data.
6. Accept the value in the **User_Name** field (sysdba) and **Password** field (masterkey).
7. To test the connection, click the button with the checkmark on it (just above the **Connection Name** list).

   Note: By default, you are prompted to log in. Use the masterkey password. If the connection works a confirmation message appears. If you cannot connect to the database, make sure you have installed Interbase and that the server is started.
8. Click **OK** to close the **Connection Editor** and save your changes.

To set up the unidirectional dataset

1. From the **dbExpress** category of the **Tool Palette**, place a TSQLDataSet component at the top of the form.
2. In the **Object Inspector**, set the SQLConnection property drop-down list to SQLConnection1.
3. Set the CommandText to a SQL command, for example, Select * from sales.

   For the SQL command, you can either type a Select statement in the **Object Inspector** or click the ellipsis to the right of CommandText to display the **Command Text Editor** where you can build your own query statement.

   Tip: If you need additional help while using the **Command Text Editor**, click the **Help** button or press F1.
4. In the **Object Inspector**, set the Active property to **True** to open the dataset.

To add the provider

1. From the **Data Access** category of the **Tool Palette**, place a TDataSetProvider component at the top of the form.

2 In the **Object Inspector**, set the DataSet property drop-down list to SQLDataSet1.

To add client dataset

1 From the **Data Access** category of the **Tool Palette**, place a TClientDataSet component to the right of the DataSetProvider component on the form.
2 In the **Object Inspector**, set the ProviderName drop-down to DataSetProvider1.
3 Set the Active property to **True** to allow data to be passed to your application.

A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1 From the **Data Access** category of the **Tool Palette**, place a TDataSource component to the right of the ClientDataSet on the form.
2 In the **Object Inspector**, set the DataSet property drop-down to ClientDataSet1.

To connect a DataGrid to the DataSet

1 From the **Data Controls** category of the **Tool Palette**, place a TDBGrid component on the form.
2 In the **Object Inspector**, set the DataSource property drop-down to DataSource1.
3 Save all files in the project.
4 Select **Run** ▸ **Run**.

   You are prompted to enter a password.
5 Enter masterkey as the password.

   The application compiles and displays a VCL.NET form with a DBGrid.

Building a VCL Forms Hello World Application

The Windows Forms *Hello World* application demonstrates the essential steps for creating a VCL Forms application. The application uses a VCL Form, a control, an event, and displays a dialog in response to a user action.

Creating the *Hello World* application consists of the following steps:

1. Create a VCL.NET Form with a button control.
2. Write the code to display "Hello World" when the button is clicked.
3. Run the application.

To create a VCL Form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
   The **VCL Forms Designer** displays.
2. Click the **Design** tab to display the form view.
3. From the **Standard** category of the **Tool Palette**, place a TButton component on the form.

To display the "Hello World" string

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick event handler on the **Events** tab.
   The **Code Editor** appears, with the cursor in the TForm1.Button1Click event handler block.
3. Place the cursor before the `begin` reserved word and then press Return.
   This creates a new line above the code block.
4. Insert the cursor on the new line created, and type the following variable declaration:

```
var s: string;
```

5. Insert the cursor within the code block, and type the following code:

```
s:= 'Hello World!';
ShowMessage(s);
```

To run the "Hello World" application

1. Save your project files.
2. Choose **Run** ▸ **Run** to build and run the application.
   The form displays with a button called **Button1**.
3. Click **Button1**.
   A dialog box displays the message "Hello World!".
4. Click **OK** to close the message dialog.
5. Close the VCL form to return to the IDE.

Building a VCL.NET Forms ADO.NET Database Application

The following procedure describes how to build an ADO.NET database application.

Building a VCL.NET ADO.NET application consists of the following major steps:

1. Set up the database connection.
2. Set up the dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add an ADO connection component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.

   The **VCL Forms Designer** displays.
2. From the **dbGO** category of the **Tool Palette**, place a TADOConnection component on the form.
3. Double-click the TADOConnection component to display the **ConnectionString** dialog.
4. If necessary, select **Use Connection String**; then click the **Build** button to display the **Link Properties** dialog box.
5. On the **Provider** page of the dialog, select **Microsoft Jet 4.0 OLE DB Provider**; then click the **Next** button to display the **Connections** page.
6. On the **Connections** page, click the ellipsis button to browse for the dbdemos.mdb database. The default path to this database is C:\Program Files\Common Files\Borland Shared\Data.
7. If it is not already filled in, enter Admin in the **User name** field and select the **Blank password** check box.
8. Click **Test Connection** to confirm the connection.

   A dialog appears, indicating the status of the connection.
9. Click **OK** twice to close the **Data Link Properties** dialog box and the **ConnectionString** dialog box.

To set up the dataset

1. From the **dbGO** category, double-click a TADODataSet component to place it on the form.
2. In the **Object Inspector**, set the Connection property drop-down list from the **Linkages** category to ADOConnection1.
3. Set the CommandText to an SQL command, for example, Select * from orders.

   You can either type the Select statement in the **Object Inspector** or click the ellipsis button to the right of CommandText to display the **Command Text Editor** where you can build your own query statement.

   Tip: If you need additional help while using the **CommandText Editor**, click the **Help** button or press F1.
4. Set the Active property to **True** to open the dataset.

   You are prompted to log in.
5. Enter Admin for the username.

6 Leave the password field blank.

To add the provider

1 From the **Data Access** category of the **Tool Palette**, double-click a TDataSetProvider component to place it at the top of the form.
2 In the **Object Inspector**, set the DataSet property to ADODataSet1.

To add client dataset

1 From the **Data Access** category of the **Tool Palette**, double-click a TClientDataSet component to place it to the right of the DataSetProvider component on the form.
2 In the **Object Inspector**, set the ProviderName property to DataSetProvider1.
3 Set the Active property to **True** to allow data to be passed to your application.

   A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1 From the **Data Access** category of the **Tool Palette**, double-click a TDataSource component to place it to the right of the ClientDataSet on the form.
2 In the **Object Inspector**, set the DataSet property to ClientDataSet1.

To connect a DataGrid to the DataSet

1 From the **Data Controls** area of the **Tool Palette**, double-click a TDBGrid component to place it on the form.
2 In the **Object Inspector**, set the DataSource property to DataSource1.
3 Select**Run** ▸ **Run**.

   You are prompted to log in.
4 Enter Admin for the username.
5 Leave the password field blank.
6 Click **OK**.

   The application compiles and displays a VCL form with a DBGrid.

Building an Application with XML Components

This example creates a VCL Forms application that uses an XMLDocument component to display contents in an XML file.

The basic steps are:

1. Create an XML document.
2. Create a VCL form.
3. Place an XMLDocument component on the form, and associate it with the XML file.
4. Create VCL components to enable the display of XML file contents.
5. Write event handlers to display XML child node contents.
6. Compile and run the application.

To create the XML document

1. Copy the text below into a file in a text editor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings [
    <!ELEMENT StockHoldings (Stock+)>
    <!ELEMENT Stock (name)>
    <!ELEMENT Stock (price)>
    <!ELEMENT Stock (symbol)>
    <!ELEMENT Stock (shares)>
]>

<StockHoldings>
  <Stock exchange="NASDAQ">
    <name>Borland</name>
    <price>10.375</price>
    <symbol>BORL</symbol>
    <shares>100</shares>
  </Stock>

  <Stock exchange="NYSE">
    <name>MyCompany</name>
    <price>8.75</price>
    <symbol>MYCO</symbol>
    <shares type="preferred">25</shares>
  </Stock>
</StockHoldings>
```

2. Save the file to your local drive as an XML document. Give it a name such as stock.xml.
3. Open the document in your browser.

   The contents should display without error.

> **Note:** In the browser, you can choose **View** ▸ **Source** to view the source file in the text editor.

To create a form with an XMLDocument component

1. Start a new project.

2 Choose **File** ▸ **New** ▸ **Other**.

3 In the **New Items** dialog box, select **Delphi for .NET Projects**.

4 Double-click **VCL Forms Application**.

The **VCL Forms Designer** displays.

5 From the **Internet** category on the **Tool Palette**, place an TXMLDocument component on the form.

6 In the **Object Inspector**, click the ellipsis button next to the FileName property, browse to the location of the XML file you created, and open it.

The XML file is associated with the TXMLDocument component.

7 In the **Object Inspector**, set the Active property to **True**.

To set up the VCL components

1 From the **Standard** page on the **Tool Palette**, place a TMemo component on the form.

2 From the **Standard** page on the **Tool Palette**, place two TButton components on the form just above **Memo1**.

3 In the **Object Inspector** with **Button1** selected, enter Borland for the Caption property.

4 In the **Object Inspector** with **Button2** selected, enter MyCompany for the Caption property.

To display child node contents in the XML file

1 Select **Button1**.

2 In the **Object Inspector** double-click the OnClick event on the **Events** tab.

The code displays with the cursor in the `TForm1.Button1Click` event handler block.

3 Enter the following code to display the stock price for the first child node when the Borland button is clicked:

```
BorlandStock:=XMLDocument1.DocumentElement.ChildNodes[0];
  Price:= BorlandStock.ChildNodes['price'].Text;
  Memo1.Text := Price;
```

4 Add a `var` section just above the code block, above the `begin statement` in the event handler, and enter the following local variable declarations:

```
var
     BorlandStock: IXMLNode;
  Price: string;
```

5 Select **Button2**.

6 In the **Object Inspector** double-click the OnClick event on the **Events** tab.

The code displays with the cursor in the `TForm1.Button2Click` event handler block.

7 Enter the following code to display the stock price for the second child node when the MyCompany button is clicked:

```
MyCompany:=XMLDocument1.DocumentElement.ChildNodes[1];
Price:= MyCompany.ChildNodes['price'].Text;
Memo1.Text := Price;
```

8 Add a `var` section just above the code block, above the begin statement in the event handler, and enter the following local variable declarations:

```
var
  MyCompany: IXMLNode;
  Price: string;
```

To compile and run the application

1 Select **Run** ▸ **Run** to compile and execute the application.

The application displays two buttons and a memo.

2 Click the **Borland** button.

The stock price displays.

3 Click the **MyCompany** button.

The stock price displays.

Building VCL Forms Applications With Graphics

Each of the procedures listed below builds a VCL Form application that uses graphics. Build one or more of the examples.

1. Draw straight lines.
2. Draw rectangles and ellipses.
3. Draw a polygon.
4. Display a bitmap image.
5. Place a bitmap in a combo box.

Creating a New VCL.NET Component

You can use the New VCL Component Wizard to create a new VCL.NET component. The wizard detects which personality of the product you are using and creates the appropriate type of component.

To create a new VCL.NET component

1. Specify an ancestor component.
2. Specify the class name.
3. Create a unit or add the unit to a package.

To specify an ancestor component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
2. Choose **Component** ▸ **New VCL Component**.

   This displays the first page of the **New VCL Component** wizard.
3. Select **VCL for Delphi.NET**.
4. Click **Next**.

   This displays the second page of the **New VCL Component** wizard and loads the page with ancestor components.
5. Select an ancestor component from the list.
6. Click **Next**.

   This displays the third page of the **New VCL Component** wizard.

To specify a class name

1. If you want to change the default class name, enter it in the **Class Name** textbox.
2. Enter the name of the area on the **Tool Palette** where you want the component to appear in the **Palette Page** textbox.
3. Enter the unit name in the **Unit Name** textbox.
4. Enter the search path in the **Search Path** textbox.
5. Click **Next**.

Note: You can also take the default values.

To create a unit

1. Select the **Create Unit** radio button.
2. Click **Finish**.

To install a unit into an existing package

1. Select the **Install into Existing Package** radio button.
2. Click **Next**.

This generates a list of existing packages.

3 Select the package you want to install the unit into.

4 Click **Finish**.

To install a unit into a new package

1 Select the **Install into New Package** radio button.

2 Click **Next**.

3 Enter a name for the package into the **File Name** textbox.

4 Enter a description for the package into the **Description** textbox.

5 Click **Finish**.

The new unit opens in the **Code Editor**.

Creating Actions in a VCL Forms Application

Using Developer Studio 2006, the following procedures illustrate how to create actions using the ActionList tool. You will set up a simple application and create an edit menu with cut and paste actions that can be used to cut and paste to a memo.

Creating the VCL application consists of the following major steps:

1. Add main menu, actionlist, and memo tools to a form.
2. Create the cut and paste actions.
3. Add the actions to the main menu and associate with the edit action category.
4. Build and run the application.

To add the main menu, actionlist, and memo to a form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delph for .NET Projects** ▸ **VCL Forms Application** to create a new form.
2. Click the **Design** tab to switch to the **VCL Form Designer**.
3. From the **Standard** category of the **Tool Palette**, place a TMainMenu, TActionList, and TMemo component on the form.

To create the actions

1. Double-click ActionList1 on the form.

   The **ActionList Editor** displays.
2. Select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog box.
3. Scroll to the TEditCut action, select it, and click **OK**.

   EditCut1 displays in the Actions list in the editor.
4. Select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog box.
5. Scroll to the TEditPaste action, select it, and click **OK**.

   EditPaste1 displays in the Actions list in the editor.
6. Close the **ActionList Editor** window.

To add the cut and paste actions to the edit category in the main menu

1. Double-click MainMenu1 on the form.

   The **MainMenu1 Editor** displays with the first blank command category selected.
2. In the **Object Inspector**, enter Edit for the Caption property and press ENTER.

   Edit displays as the first command category.
3. Click **Edit** to display a blank action just below it.
4. Click the blank action to select it.
5. In the **Object Inspector**, select **EditCut1** from the drop-down list of actions in the Action property, located in the **Linkage** category.
6. If not already filled in, expand the list of Action properties, enter Cut for the Caption property, enter Edit for the category, and press ENTER.

   Cut displays as the first action.

7 In the **MainMenu1 Editor**, click the second blank action beneath **Cut** to select it.

8 In the **Object Inspector**, select **EditPaste** from the drop-down list of actions in the Action property, located in the **Linkage** category.

9 Expand the list of Action properties, and if necessary, enter Paste for the Caption property, enter Edit for the category, and press ENTER.

Paste displays as the second action.

To build and run the application

1 Save all files in the project.

2 Choose **Run** ▸ **Run**.

The application executes, displaying a form with the main menu bar and the **Edit** menu.

3 In the application, select text in the memo.

4 Choose **Edit** ▸ **Cut**.

The text is cut from the memo.

5 Choose **Edit** ▸ **Paste**.

The text is pasted back into the memo.

Displaying a Bitmap Image in a VCL Forms Application

These procedures load a bitmap image from a file and displays it to a VCL form.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's onClick event handler to load and display a bitmap image.
4. Build and run the application.

To create a VCL form and button

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
   The **VCL Forms Designer** displays.
2. From the **Standard** category in the **Tool Palette**, place a TButton component on the form.

To provide a bitmap image

1. Create a directory in which to store your project files.
2. Locate a bitmap image and copy it to your project directory.
3. Save all files in your project to your project directory.

To write the OnClick event handler

1. In the **Input** category of the **Events** tab, double-click the Button1OnClick event.
   The **Code Editor** displays with the cursor in the `TForm1.Button1Click` event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with the name of the bitmap image in your project directory:

```
            Rect := TRect.Create(0,0,100,100);
  Bitmap := TBitmap.Create;
    try
      Bitmap.LoadFromFile('MyFile.bmp');
      Form1.Canvas.Brush.Bitmap := Bitmap;
      Form1.Canvas.FillRect(Rect);
    finally
      Form1.Canvas.Brush.Bitmap := nil;
      Bitmap.Free;
    end;
```

   Tip: You can change the size of the rectangle to be displayed by adjusting the `Rect` parameter values.

3. In the var section of the code, add these variable declarations:

```
Bitmap : TBitmap;
Rect : TRect;
```

To run the program

1. Save all files in your project.
2. Choose **Run** ▸ **Run**.
3. Click the button to display the image bitmap in a 100 x 100-pixel rectangle in the upper left corner of the form.

Drawing a Rounded Rectangle in a VCL Forms Application

These procedures draw a rounded rectangle in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a polygon.
3. Build and run the application.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
2. In the **Designer**, click the form, if necessary, to display Form1 properties in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, click the **Events** tab.
2. Double-click the OnPaint event handler in the **Visual** category.
   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` event handler block.
3. Enter the following event handling code:

```
Canvas.RoundRect(0, 0, ClientWidth div 2,
  ClientHeight div 2, 10, 10);
```

To run the program

1. Save all files in your project.
2. Select **Run** ▸ **Run**.
3. The application executes, displaying a rounded rectangle in the upper left quadrant of the form.

Drawing Rectangles and Ellipses in a VCL Forms Application

These procedures draw a rectangle and ellipse in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a rectangle and ellipse.
3. Build and run the application.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
2. In the **Object Inspector**, click the **Design** tab, if necessary, to display Form1.

To write the OnPaint event handler

1. In the **Object Inspector**, click the **Events** tab.
2. Double-click the OnPaint event in the **Miscellaneous** category on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` event handler block.
3. Enter the following event handling code:

```
Canvas.Rectangle (0, 0, ClientWidth div 2, ClientHeight div 2);
Canvas.Ellipse (0, 0, ClientWidth div 2, ClientHeight div 2);
```

To run the program

1. Save all files in your project.
2. Choose **Run** ▸ **Run**.
3. The application executes, displaying a rectangle in the upper left quadrant, with an ellipse in the middle of the rectangle.

Drawing Straight Lines In a VCL Forms Application

These procedures draw two diagonal straight lines on an image in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw the straight lines.
3. Build and run the application.

To create a VCL form and place an image on it

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
2. Click the **Design** tab, if necessary, to display Form1 properties in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, click the **Events** tab.
2. In the **Visual** category, double-click the OnPaint event.
   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` event handler block.
3. Enter the following event handling code:

```
with Canvas do
        begin
                MoveTo(0,0);
                LineTo(ClientWidth, ClientHeight);
                MoveTo(0, ClientHeight);
                LineTo(ClientWidth, 0);
        end;
```

To run the program

1. Save all files in your project.
2. Choose **Run** ▸ **Run**.
   The application executes, displaying a form with two diagonal crossing lines.

Tip: To change the color of the pen to green, insert this statement following the first `MoveTo()` statement in the event handler code: `Pen.Color := clRed;` Experiment using other canvas and pen object properties.

Importing .NET Controls to VCL.NET

You might want to use .NET components on your VCL.NET forms. There is no direct way to use .NET components. You can, however, wrap the components in an ActiveX wrapper, which then can be added to your VCL.NET application. Developer Studio 2006 provides the **.NET Import Wizard** to accomplish this task.

To use .NET components in a VCL.NET Form

1. Run the WinForm Control Import Wizard.
2. Build the package to create an assembly file.
3. Add the assembly to the **Tool Palette**.

To run the WinForm Control Import Wizard

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **WinForm Controls Package**.

   This starts the **WinForm Control Import Wizard**.
2. Specify the following file:

   ```
   c:\Windows\Microsoft.NET\Framework\v1.1.4322\System.Windows.Forms.dll
   ```
3. Click **Next**.

   This displays the second page of the **WinForm Control Import Wizard** and lists all of the available components.
4. Check the check boxes next to the components you want to import.

   Note: If you want to import all components, click the **Check All** button.
5. Click **Next**.

   This displays the third page of the Wizard, which provides generation options for the units.
6. Accept the defaults, and click **Next**.

   This displays the fourth page of the Wizard, which allows you to set a location and a name for the package file.
7. Click **Next**.

   This displays the fifth page of the Wizard, which allows you to overwrite any existing files of the same name.
8. Click **Next**.

   This initiates the generation process and displays status messages for each file as it is created, including the package (.dpk) file.
9. If you want to import additional controls, click **New**. Otherwise, click **Finish**.

   The package containing the units appears in the **Project Manager**.

To build and add the package

1. Select the package node in the **Project Manager**.
2. Choose **Project** ▸ **Build <Project Name>** from the main menu where <Project Name> is the name of your project.

   This creates the assembly file containing the package and the units.
3. Choose **Components** ▸ **Installed .NET Components**.

4 Click the **.NET VCL Components** tab.

5 Click **Add**.

6 Locate the package assembly, select it, and click **Open**.

The location depends on your project options directory locations. The file might also end up in your default documents directory.

7 Click **OK**.

The individual controls appear in the **Tool Palette** under the **WinForm Controls** category. You can now add the individual controls to your VCL.NET form applications.

Placing a Bitmap Image in a Control in a VCL Forms Application

These procedures add a bitmap image to a combo box in a VCL forms application.

1. Create a VCL form.
2. Place components on the form.
3. Set component properties in the Object Inspector.
4. Write event handlers for the component's drawing action.
5. Build and run the application.

To create a VCL form with a ComboBox component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application**.
   The **VCL Forms Designer** displays.
2. Click the **Design** tab to display the form.
3. From the **Win32** category of the **Tool Palette**, place an TImageList component on the form.
4. From the **Standard** category of the **Tool Palette**, place a TComboBox component on the form.

To set the component properties

1. Select ComboBox1 on the form.
2. In the **Object Inspector**, set the Style property drop-down to csOwnerDrawFixed.
3. In the **Object Inspector**, click the **[...]** next to the Items property.
   The **String List Editor** displays.
4. Enter a string you would like to associate with the bitmap image, for example, MyImage and then click **OK**.
5. Double-click ImageList1 in the form.
   The **ImageList Editor** displays.
6. Click the **Add** button to display the **Add Images** dialog.
7. Browse your local drive to locate a bitmap image to display in the combobox.
8. Select a very small image such as an icon. Copy it to your project directory, and click **Open**.
   The image displays in the **ImageList Editor**.
9. Click **OK** to close the editor.

To add the event handler code

1. In the **Designer**, select ComboBox1.
2. In the **Object Inspector**, click the **Events** tab.
3. Double-click the OnDrawItem event.
   The **Code Editor** displays with cursor in the code block of the DrawItem event handler.
4. Enter the following code for the event handler:

```
ComboBox1.Canvas.FillRect(rect);
ImageList1.Draw(ComboBox1.Canvas, Rect.Left, Rect.Top, Index);
```

```
ComboBox1.Canvas.TextOut(Rect.Left+ImageList1.Width+2,
  Rect.Top, ComboBox1.Items[Index]);
```

To run the program

1. Save all files in your project.
2. Choose **Run** ▸ **Run**.

   The application executes, displaying a form with a combo box.
3. Click the combobox drop-down.

   The bitmap image and the text string display as a list item.

Using ActionManager to Create Actions in a VCL Forms Application

Using Developer Studio 2006, the following procedure illustrates how to create actions using ActionManager. It sets up a simple user interface with a text area, as would be appropriate for a text editing application, and describes how to create a file menu item with a file open action.

Building the VCL application with ActionManager actions consists of the following major steps:

1. Add a file open action to the ActionManager on a form.
2. Create the main menu.
3. Add the action to the menu.
4. Build and run the application.

To add a file open action to ActionManager

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **VCL Forms Application** to create a new form.
2. From the **Additional** page of the **Tool Palette**, add a TActionManager component to the form.
3. Double-click the TActionManager component to display the **Action Manager** editor.

   Tip: To display captions for nonvisual components such as TActionManager, choose **Tools** ▸ **Environment Options**. On the **Designer** tab, check **Show component captions**, and click **OK**.

4. If necessary, click the **Actions** tab.
5. Select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog.
6. Scroll to the **File** category, and click the TFileOpen action.
7. Click **OK** to close the dialog.
8. In the **Action Manager** editor, select the **File** category.

   Open... displays in the **Actions:** list box.
9. Click **Close** to close the editor.

To create the main menu and add the File action to it

1. From the **Additional** page of the **Tool Palette**, place a TActionMainMenuBar component on the form.
2. Open the **Action Manager** editor, and select the **File** category from the **Categories** list box.
3. Drag **File** to the blank menu bar.

   File displays on the menu bar.
4. Click **Close** to close the editor.

To build and run the application

1. Select **Run** ▸ **Run**.

   The application executes, displaying a form with the main menu bar and the **File** menu.
2. Select **File** ▸ **Open**.

   The **Open** file dialog displays.

Web Services Procedures

Accessing an ASP.NET "Hello World" Web Services Application

If you want to consume the Web Services application you created, you must create a client application to access your ASP.NET Web Services application. This process requires different development steps to achieve the desired output.

To access a simple "Hello World" ASP.NET Web Services application

1. Create a client application.
2. Add a Web Reference for an XML web service.
3. Create the code-behind logic.
4. Run the client application.

To create a client application

1. Choose **File** ▸ **New** ▸ **Other**.

   A **New Items** dialog box appears.
2. Select any type of application to create your client, such as a Windows Forms application or an ASP.NET Web application.

   For this example, we will create a Windows Forms application (either Delphi for .NET or C#).
3. Click **OK**.

   A **New Project** dialog box appears.

To add a Web Reference for an ASP.NET Web Services application

1. Choose **Project** ▸ **Add Web Reference**.
2. From the **Borland UDDI Browser** web dialog box, enter the following URL in the address text box at the top:

   ```
   http://localhost/WebService1/WebService1.asmx
   ```

   Note: The name of your application may not be WebService1. In that case, use your own application name in place of WebService1 in the example preceding example.
3. Press Enter.

   Note: If you need to determine the proper path and you are using IIS, you can open the IIS Administrator from the Windows XP Control Panel Administrative Tools. Find the WebService you have saved and compiled in the list of IIS web sites, then review the name of the site and the name of the .asmx file.

   If you have entered the proper path, this should display information about the WebMethods.
4. Click the **Service Description** link to view the WSDL document.
5. Click **Add Reference** to add the WSDL document to the client application.

   A **Web References** folder is added to the Project directory in the **Project Manager** which contains the WebService1.wsdl file and the dialog box disappears.

To create the code-behind logic

1. Add a Button to the Windows Form.
2. Double-click the Button to view the code-behind file.
3. For a Delphi for .NET client, implement the Click event in the **Code Editor** with the following code:

```
[Delphi]
procedure TWinForm.Button1_Click(sender: System.Object; e: System.EventArgs);
var
    ws: TWebService1;
begin
                ws := TWebService1.Create;
                button1.Text := ws.HelloWorld();
end;
```

   When you added the Web Reference to your application, Developer Studio 2006 used the WSDL to generate a proxy class representing the "Hello World" web service. The Click event uses methods from the proxy class to access the web service. For a Delphi for .NET client, you may need to add the unit name of the proxy class (for example, localhost.WebService1) to the uses clause of your Windows Form unit to prevent errors in your Click event.

4. For a C# client, implement the Click event in the **Code Editor** with the following code:

```
[C#]

    private void button1_Click(object sender, System.EventArgs e)
        {
                  TWebService1 ws = new TWebService1();
                  button1.Text = ws.HelloWorld();
        }
```

To run the client application

1. Save the application.
2. Compile and run the project.
3. Click the Button on your client application.

   The "Hello World" caption appears on the button.

Adding Web References in ASP.NET Projects

If you want to consume a web service, you must create a client application, and add a Web Reference. These procedures describe how to create an ASP.NET client application that consumes a third-party web service. The client application consumes the DeadOrAliveWS web service available from the XMethods Web site. This web service lets you query a simple database of celebrities and their respective birthdates and expiration dates.

To create an ASP.NET project

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box appears.
2. Double-click the **ASP.NET Web Application** icon in either the **C# Projects** or **Delphi for .NET Projects** item categories.

   The **New ASP.NET Application** dialog box appears.
3. In the **Name** field, enter a name for your project.
4. In the **Location** field, enter a path for your project.

   Tip: Most ASP.NET projects reside in the IIS directory Inetpub\wwwroot.
5. If necessary, click the **View Server Options** button to change your Web server settings.

   Tip: The default Server Options will usually be sufficient, so this step is optional.
6. Click **OK**.

   The Web Forms Designer appears.

To design the ASP.NET web page

1. If necessary, click **Design** view.
2. From the **Web Controls** category of the **Tool Palette**, place a Button component onto the Designer surface.

   The Button control appears on the Designer. Make sure the control is selected.
3. In **Object Inspector**, set the **Text** property to Dead or Alive?.
4. From the **Web Controls** category of the **Tool Palette**, place a TextBox component onto the Designer above the Button.

   This is where you type your query to the Web Service.
5. Place a Label component below the Button.

   This is where the results of the web service query are displayed.

Use the UDDI browser to locate the DeadOrAlive Web Service on the internet. This allows you to use the methods and objects published by the Web Service Definition Language (WSDL).

To add the Web Reference for DeadOrAliveWS

1. Choose **Project** ▸ **Add Web Reference**.
2. In the **Borland UDDI Browser** web dialog box, click the **XMethods Full** link in the list of available UDDI directories.

   A list of various web services published on the XMethods Web site appears.

3 Find and click the **DeadOrAliveWS** link.

Tip: You can use Ctrl+F to search within the **Borland UDDI Browser**.

4 Click the link to the WSDL file:

```
http://www.abundanttech.com/webservices/deadoralive/deadoralive.wsdl
```

A WSDL document appears. This XML document describes the interface to the DeadOrAliveWS web service.

5 Click **Add Reference** to add the WSDL document to the client application.

A **Web References** folder containing a **com.abundanttech.www** node is added to the Project directory in the **Project Manager**.

To write the application logic

1 If necessary, click **Design** view.

2 Double-click the **Dead or Alive?** button to view the code-behind file.

3 For a Delphi for .NET Web Services application, implement the Click event in the **Code Editor** with the following code :

```
[Delphi]
procedure TWebForm1.Button1_Click(sender: System.Object; e: System.EventArgs);
var
  result: DataSet;
  ws: DeadOrAlive;
  currentTable: DataTable;
  currentRow: DataRow;
  currentCol: DataColumn;
begin
  //This initializes the web service
  ws := DeadOrAlive.Create;

  //Send input to the web service
  result := ws.getDeadOrAlive(TextBox1.Text);

  //parse results and display them
  Label1.Text := '';
  for currentTable in result.Tables do
    begin
      Label1.Text := Label1.Text + '<p>' + #13#10;
      for currentRow in currentTable.Rows do
        begin
          for currentCol in currentTable.Columns do
            begin
              Label1.Text := Label1.Text + currentCol.ColumnName + ': ';
              Label1.Text := Label1.Text + (currentRow[currentCol]).ToString;
              Label1.Text := Label1.Text  + '<br>' + #13#10;
            end;
        end;
      Label1.Text := Label1.Text + '</p>';
    end;

end;
```

When you added the Web Reference to your application, Developer Studio 2006 used the WSDL to generate a proxy class representing the "Hello World" web service. The Click event uses methods from the proxy class to access the web service. For Delphi for .NET Web Services, you may need to add the unit name of the proxy class, abundanttech.deadoralive, to the uses clause of your Web Form unit to prevent errors in your Click event.

4 For a C# Web Services application, implement the Click event in the **Code Editor** with the following code :

```
[C#]
private void button1_Click(object sender, System.EventArgs e)
{
        DataSet result;

        //This initializes the web service
        DeadOrAlive source = new DeadOrAlive();

        //Send input to the web service
        result = source.getDeadOrAlive(textBox1.Text);

        //parse results and display them
        label1.Text = "";
        foreach (DataTable currentTable in result.Tables) {
                label1.Text += "<p>\n";
                foreach (DataRow currentRow in currentTable.Rows) {
                        foreach (DataColumn currentCol in currentTable.Columns) {
                                label1.Text += currentCol.ColumnName + ": ";
                                label1.Text += currentRow[currentCol] + "<br>\n";
                        }
                }
            label1.Text += "</p>";
        }
}
```

Note: As you can see by the added application logic code, the DeadOrAliveWS web service returns query results in the form of a dataset. Web Services can, however, return data in a variety of formats.

To run the application

1 Choose **Project** ▸ **Build All Projects**.

Now your project is built and resides on your ASP.NET server.

2 Open a Web browser.

3 Type the URL of your Web Application's .aspx file and press Enter.

Tip: If you are using Microsoft IIS, the URL is the path of the .aspx file after Inetpub\wwwroot. For example, if the path of your Web Application is c:\Inetpub\wwwroot\WebApplication1 and your .aspx file is named "WebForm1.aspx", the URL would be http://localhost/WebApplication1/WebForm1.aspx.

4 If necessary, enter your user name and password for your ASP.NET server.

The web page for your web application appears.

5 Enter the name of a celebrity (for example, Isaac Asimov) in the text box and click the **Dead or Alive?** button.

Your web application requests the information from the DeadOrAliveWS web service and displays the result in the label.

Note: If no information is displayed, that name may not be in the database. Check your spelling or try a different name.

Building an ASP.NET "Hello World" Web Services Application

Building an application with ASP.NET Web Services lets you expose functionality to your client application over a Web connection. These steps walk you through building a simple "Hello World" application with ASP.NET Web Services. Once built, the application exposes all of its objects and methods through a WebMethod that you create and access through a web browser.

To create a simple "Hello World" application with ASP.NET Web Services

1. Create an ASP.NET Web Services application.
2. Create a WebMethod.
3. Test and run the ASP.NET Web Services application.

Note: Currently, using Developer Studio 2006 you can only create web services using the code-behind method. You cannot use the code inline method, in which you code your web service in the <ServiceName>.asmx file. Currently, Developer Studio 2006 does not support the code inline method of creating web services.

To create an an ASP.NET Web Services application

1. Choose **File** ▸ **New** ▸ **Other**.

   A **New Items** dialog box appears.
2. Select the **ASP Projects** folder for the language you are using.
3. Select **ASP.NET Web Service Application**.

   An **Application Name** dialog box appears.
4. Enter a name and location of the application in the fields and retain all other default settings.

   Note: If you are using the Cassini Web Server, you need to change the Location and Server entries. You also need to make sure you configure the Cassini Web Server before trying to run this application. Choose **Tools** ▸ **Options** and select **ASP.NET Options** to set the **Path** and the **Port** for Cassini.
5. Click **OK**.

   A WebService1.asmx file and a WebService1.asmx.<filetype>, are automatically created for you.

To create a WebMethod

1. Select the **WebService.pas** or **WebService.asmx.cs** tab at the bottom of the **Code Editor**.

   If you named your web service application something other than the default, that will be the name that appears on the tab. The code for the "Hello World" application is already included in the WebMethod that is created for you when you created the Web Services application.
2. Uncomment the sample WebMethods in the code-behind file.

   Delphi for .NET applications have two "Hello World" WebMethods to uncomment; one in the Interface module and the other in the Implementation module.

   In C# Web Services applications, uncomment the "HelloWorld" and "EchoString" WebMethods.
3. Choose **Project** ▸ **Build <project name>** to build your project.
4. Run your project.

   This invokes the browser which hosts the Web Service.

The pages you see will include sample SOAP and HTTP code that you can use to access the WebMethods. You can run the samples and see how the results are passed to an output XML file.

To test and run the XML web service manually

1 From a web browser, enter the location of the WebService1.asmx file on your localhost:

```
http://localhost/WebService1/WebService1.asmx
```

The pages you see will include sample SOAP and HTTP code that you can use to access the WebMethods. You can run the samples and see how the results are passed to an output XML file.

> **Note:** You may need to use a slightly different syntax than that shown in this step. For instance, on some Windows XP machines, the *localhost* identifier should be your machine name. For instance, if you machine name is *MyMachine*, the syntax would be: `http://MyMachine/WebService1/Webservice1.asmx`.

2 Test the two methods from a web browser.

Porting a Delphi for Win32 Web Service Client Application to Delphi for .NET

The following steps are required to port your Delphi for Win32 Web Services client application to Delphi for .NET.

To port your web service

1. Change the existing RIO form components.
2. Change the uses clause.
3. Add a web reference.
4. Change the web service invocation code.

To change your existing form components

1. Copy and save the web reference URL from your existing RIO component.
2. Delete the HTTPRio component from the form if it was not dynamically created.

To change the uses clause

1. Remove any Delphi for Win32 SOAP units from the clause.

   These include, but are not restricted to InvokeRegistry, RIO, and SOAPHTTPClient.

   Warning: The preceding list of units is not inclusive. Make sure you identify all SOAP units, regardless of naming convention. Not all of the units include the word SOAP in the name.

2. Remove the reference to the Delphi for Win32 WSDL Importer-generated Interface proxy unit.
3. Remove the proxy unit from the project.

To add a web reference

1. Open a Delphi for Win32 project in Developer Studio 2006 and choose **Project** ▸ **Add Web Reference**.

   Once you have saved the project, the UDDI Browser appears.

2. Enter the URL you want to use, either a service you are already familiar with, or the one saved from your RIO component into the list box at the top of the Browser.

   Note: If you want to locate a WSDL file on your local disk, you can click the ellipsis button next to the list box and search for the document. You can also navigate to one of the web service sites listed in the UDDI Browser if you want to use a published service.

3. Click the **Add Reference** button to add the WSDL document to your project.

   Developer Studio 2006 creates the necessary web reference and the corresponding proxy unit based on the WSDL document. A new Web References node appears in the **Project Manager**. Expand it to see the associated WSDL and proxy code files.

4. Choose **File** ▸ **Use Unit**.

To change the web service invocation code

1 In the code file for your application, locate the code that invokes the web service.

Assume it looks something like this:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
    HelloService: Service3Soap;
begin

    // The next line will be slightly different if you have
    // used a component or generated the method dynamically.

 // This is how it will look if you used a component.
    HelloService := (HTTPRIO1 as Service3Soap);

    // This is how it will look if created dynamically.
    // GetService3Soap is the global method in the proxy unit.
 HelloService := GetService3Soap;

    Caption := HelloService.HelloWorld;
end;
```

2 Change the var section from this:

```
[Delphi]
var
// This is the type of the old proxy interface.
    HelloService: Service3Soap;
```

to

```
[Delphi]
var
// This is the type of the new proxy class.
    HelloService: Service3;
```

This assumes the name of your service is Service3. Change the name accordingly.

> **Note:** You will see that what was formerly created as an interface is now created as a class. The .NET Framework provides automatic garbage collection, and so certain restrictions placed on the use of classes in previous versions of Delphi may no longer apply when using Developer Studio 2006.

3 Change the first line in the procedure block from this:

```
[Delphi]
HelloService := (HTTPRIO1 as Service3Soap);
```

to:

```
[Delphi]
HelloService := Service3.Create;
```

The updated code should look like this:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
    HelloService: Service3;
begin
    HelloService := Service3.Create;
    Caption := HelloService.HelloWorld;
end;
```

Your code is most likely more complex than this example. However, these instructions cover the basic steps for porting any Delphi for Win32 application that uses web services to Developer Studio 2006.

Windows Forms Procedures

Building a Windows Forms Application

The following procedures illustrate the essential steps to build a Windows Forms application using Developer Studio 2006.

To create a Windows Forms project

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **Windows Forms Application**.
   The **Windows Forms Designer** appears.
2. From the **Tool Palette**, place components onto the **Designer** to create the user interface.
3. Associate logic with the controls.

To associate code with a control

1. In the **Designer**, double-click a component.
   The **Code Editor** appears, cursor in place between the reserved words begin and end in the event handler.
2. Code your application logic.
3. Save and compile the application.

Building a Windows Forms Database Application

The following procedure describes the minimum number of steps required to build a simple ADO.NET application using Windows Forms and BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of Windows Forms, ADO.NET, and BDP.NET at work. To instantiate and configure a data provider, you can also drag and drop objects from the **Data Explorer**, which is a tabbed window on the right-hand side of the IDE.

Building a BDP.NET project consists of three major steps:

1. Configure BDP.NET connection components and a data source.
2. Create and configure a BdpDataAdapter.
3. Connect a DataGrid to the connection components.

To configure connection components and a data source

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.
2. Drag and drop a BdpConnection component onto the **Designer**.

   The BdpDataAdapter, BdpConnection, and other BDP.NET components can be found on the **Tool Palette** in the **Borland Data Provider** area.
3. At the bottom of the **Object Inspector**, click the **Designer Verb Connection Editor**.

   Note: Designer verbs are action phrases that appear in the lower left-hand corner of the **Object Inspector**. When you move the cursor over the phrase, the cursor changes to a hand pointer.
4. Click **Add** to add a new connection.
5. Choose a provider type from the **Provider Name** drop down list box.
6. Type the name of the provider.
7. Click **OK**.
8. Set up the connection.
9. Click **OK**.

Tip: Alternatively, use Data Explorer to drag and drop a table on to the designer surface. Data Explorer sets the connection string automatically.

To set up a connection

1. Click the **Connections Editor Designer Verb** at the bottom of the **Object Inspector**.
2. In the **Borland Data Provider: Connections Editor** dialog box, select an existing connection from the **Connections** list or add a new connection.
3. In **Connection Settings**, enter the **Database** path.

Tip: If using Interbase, you would enter the path to your Interbase database, which may be located locally in `c:\Program Files\Common Files\Borland Shared\Data`. If connecting to a shared network location, you will need to enter the network path and you will need to have access rights for that remote server.

4 Complete the **UserName** and **Password** fields for the database as needed.

Tip: If you are using a sample Interbase database, the username and password are, respectively, `sysdba` and `masterkey`.

5 Click **Test** to confirm the connection.

A dialog appears indicating the status of the connection.

6 Click **OK**.

To create and configure a data adapter

1 From the **Tool Palette**, drag and drop a BdpDataAdapter component onto the **Designer**.

2 In the **Object Inspector**, expand the **SelectCommand** property in the **Fill** area.

3 Select the connection object from the **Connection** property drop down list box.

4 Click the **Configure Data Adapter** designer verb.

This displays the **Data Adapter Configuration** editor.

5 On the **Command** tab, select a table from the **Tables** list.

6 Select one or more columns from the **Columns** list.

7 Click **Generate SQL**.

To create a dataset

1 To make sure you get the data you want, click the **Preview Data** tab on the **Data Adapter Configuration** editor.

2 Click **Refresh**.

Column and row data should appear. If they don't appear, it may be that you either do not have a live connection to a database or your SQL statement is incorrect.

3 Click the **DataSet** tab.

4 Click **New DataSet**.

5 Either accept the default name or enter a more descriptive name.

6 Click **OK**.

A new DataSet component appears in the **Component Tray** at the bottom of the IDE.

To connect a DataGrid to a DataSet

1 In the **Component Tray**, select the BdpDataAdapter.

2 In the **Live Data** area of the **Object Inspector**, set the **Active** property to **True**.

3 Drag and drop a DataGrid component from the **Data Controls** area of the **Tool Palette** onto the **Designer**. If necessary, select the DataGrid object.

4 In the **Object Inspector**, select the DataSource property drop-down from the **Data** area.

5 Select the DataSet component that you generated previously (the default is dataSet1).

6 In the **Object Inspector**, select the DataMember property drop-down.

7 Select the appropriate table.

   The DataGrid displays data from the DataSet.

8 Choose **Run** ▸ **Run**.

   The application compiles and displays a Windows Form with DataGrid.

While presenting a minimum number of steps required to build an ADO.NET project, the preceding procedure demonstrates the major components of the Windows Forms, ADO.NET, and BDP.NET architectures at work, including: connections, datasets, and adapters. The adapter connects to the physical data source by way of a connection, sending a command that reads data from the data source and populates a dataset. Once populated, a datagrid displays data from the dataset.

Alternatively, use the **Data Explorer** to create and manage database connections.

Building a Windows Forms Hello World Application

Though simple, the Windows Forms *Hello World* application demonstrates the essential steps for creating a Windows Forms application. The application uses a Windows Form, a control, an event, and will display a dialog in response to a user action.

Creating the *Hello World* application consists of three major steps:

1. Create a Windows Form.
2. Create the logic.
3. Run the application.

To create a Windows Form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi for .NET Projects** ▸ **Windows Forms Application**.
   The **Windows Forms Designer** appears.
2. From the Windows Forms **Tool Palette**, drag and drop a Button control onto the **Designer**.
3. If necessary, select the control.
4. Set the **Text** property to Hello, world!

To associate code with the button control

1. In the **Designer**, double-click the Button control.
   The **Code Editor** appears, cursor in place within the event handler code block.
2. Code the application logic:

```
MessageBox.Show('Hello, Developer!');
```

3. Save and compile the application.

To run the "Hello World" application

1. Choose **Run** ▸ **Run**.
   The application compiles and displays a Windows Form with the "Hello, world" button.
2. Click the "Hello, world!" button.
   The "Hello, developer!" dialog appears.
3. Close the Windows Form to return to the IDE.

Building Windows Forms Menus

Using Developer Studio 2006 designers, the following procedures illustrate how to create a Windows Forms context or main menu, add event handlers, and use common keyboard and pop-up options. For more information regarding the ContextMenu and MainMenu classes, see the .NET Framework Class Library.

To create a menu

1 From the **Tool Palette**, place a ContextMenu or a MainMenu component on the **Windows Forms Designer**.

A visual representation of the menu appears on the **Designer**.

> **Note:** For convenience, the ContextMenu appears much like a MainMenu component when placed on the designer.

2 Select and replace sample menu text.

When you select menu text, additional options appear for submenus and menu items. Complete as needed.

To create an event handler for a menu item

1 In the **Designer**, double-click a menu item.

The **Code Editor** appears, cursor in place between event handler brackets.

2 Code your menu item logic.

3 Save and compile the application.

You can use Arrow, Shift, and other keys to manipulate menu items.

To use keyboard sequences for menus

1 Add a MainMenu or a ContextMenu to the **Designer**.

2 Refer to the following table for keyboard sequences.

Key	Description
Delete	Removes the currently selected menu item.
Insert	Inserts a blank menu item before the currently selected menu item.
Enter	The **Designer** goes into editing mode on the currently selected item. If the **Designer** is already in editing mode, the next item becomes the currently selected item.
Arrow	Changes the currently selected menu item to the next item in the arrow direction.

You can right-click on a context menu to view a shortcut menu.

To use shortcut menus

1 Add a MainMenu or a ContextMenu to the **Designer**.

2 Right click on the menu object.

3 Refer to the following table for menu selections.

Menu Item	Description
Cut	Removes the currently selected menu item and places it on the clipboard.
Copy	Places the currently selected menu item in the clipboard.
Paste	Places a menu item from the clipboard above the currently selected menu item.
Delete	Removes the currently selected menu item.
Insert New	Inserts a blank menu item before the currently selected menu item.
Insert Separator	Inserts a separator before the currently selected menu item.
Show Code	Goes to the code and generates an event handler if one does not already exist.

Passing Parameters in a Database Application

The following procedures describe a simple application that allows you to pass a parameter value at runtime to a DataSet. Parameters allow you to create applications at design time without knowing specifically what data the user will enter at runtime. This example process assumes that you already have your sample Interbase Employee database set up and connected. For purposes of illustration, this example uses the default connector IBConn1, which is set to a standard location. Your database location may differ.

To pass a parameter

1. Create a data adapter and connection to the Interbase employee.gdb database.
2. Add a text box control, a button control, and a data grid control to your form.
3. Configure the data adapter.
4. To add a parameter to the data adapter.
5. Configure the data grid.
6. Add code to the button Click event..
7. Compile and run the application.

To create a data adapter and connection

1. Choose **File** ▸ **New** ▸ **Windows Forms Application** for either Delphi for .NET or C#.

   The Windows Forms designer appears.
2. Click on the **Data Explorer** tab and drill down to find the IBConn1 connection under the Interbase node.
3. Drag and drop the EMPLOYEE table onto the Windows Form.

   This creates a BdpDataAdapter and BdpConnection and displays their icons in the **Component Tray**.
4. Select the data adapter icon, then click the **Configure Data Adapter** designer verb in the **Designer Verb** area at the bottom of the Object Inspector.

   This displays the **Data Adapter Configuration** dialog.
5. Rewrite the SQL statement that is displayed in the Select tab of the dialog to:

   ```
   SELECT EMP_NO, FIRST_NAME, LAST_NAME, SALARY FROM EMPLOYEE WHERE FIRST_NAME = ?;
   ```

   As you can see, this statement is limiting the number of fields. It also contains a **?** character as part of the Where clause. The **?** character is a wildcard that represents the parameter value that your application passes in at runtime. There are at least two reasons for using a parameter in this way. The first reason is to make the application capable of retrieving numerous instances of the data in the selected columns, while using a different value to satisfy the condition. The second reason is that you may not know the actual values at design time. You can imagine how limited the application might be if we retrieved only data where `FIRST_NAME = 'Bob'`.
6. Click the **DataSet** tab.
7. Click **New DataSet**.
8. Click **OK**.

   This creates the DataSet that represents your query.

To add a parameter to the data adapter

1. Select the data adapter icon, then expand the properties under SelectCommand in the **Fill** area of the **Object Inspector**.

   You should be able to see your Select statement in the SelectCommand property drop down list box.
2. Change the ParameterCount property to 1.
3. Click the (Collection) entry next to the Parameters property.

   This displays the **BdpParameter Collection Editor**.
4. Click **Add** to add a new parameter.
5. Rename the parameter to *emp*.
6. Set BdpType to *String*, DbType to *Object*, Direction to *Input*, Source Column to *FIRST_NAME*, and ParameterName to *emp*.
7. Click **OK**.
8. In the **Object Inspector**, set the Active property under Live Data to **True**.

To add controls to the form

1. Drag and drop a TextBox control onto the form.
2. Drag and drop a Button onto the form.
3. Change the Text property of the button to *Get Info*.
4. Drag and drop a DataGrid data control onto the form.
5. Arrange the controls how you want them to appear, making sure that the DataGrid is long enough to display four fields of data.

To configure the data grid

1. Select the data grid.
2. Set the DataSource property to the name of the DataSet (dataSet1 by default).
3. Set the DataMember property to *Table1*.

   This should display the column names of the columns specified in the SQL statement that you entered into the data adapter.

To add code to the button Click event

1. Double-click the button to open the Code Editor.
2. In the button1_Click event code block, add the following code:

```
[C#]
bdpSelectCommand1.Close();
/* This closes the command to make sure that we will pass the parameter to */
/* the most current bdpSelectCommand.

            bdpDataAdapter1.Active = false;
/* This clears the data adapter so that we don't maintain old data

            bdpSelectCommand1.Parameters["emp"].Value = textBox1.Text;
/* This sets the parameter value to whatever value is in the text field.   */
```

```
            bdpDataAdapter1.Active = true;
/* This re-activates the data adapter so the refreshed data appears in the data grid. */
```

```
[Delphi]

Self.bdpSelectCommand1.Close();
/* This closes the command to make sure that we will pass the parameter to */
/* the most current bdpSelectCommand.

            Self.BdpDataAdapter1.Active := false;
/* This clears the data adapter so that we don't maintain old data

            Self.bdpSelectCommand1.Parameters['emp'].Value := textBox1.Text;
/* This sets the parameter value to whatever value is in the text field.   */

            Self.BdpDataAdapter1.Active := true;
/* This re-activates the data adapter so the refreshed data appears in the data grid. */
```

If you have changed the names of any of these items, you need to update these commands to reflect the new names.

3 Save your application.

To compile and run the application

1 Press Shift + F9 to compile the application.

2 Press F9 to run the application.

3 Type one of the names John, Robert, Roger, Kim, Terri, Katherine, or Ann into the text box.

4 Click the button.

This displays the employee number, first name, last name, and salary of the employee with that name in the data grid. If there is more than one person with the same first name, the grid displays all occurrences of employees with that name.

Concepts

Win32

Building Windows Applications with Win32 Forms

Windows provides a traditional approach to developing user interfaces, client/server applications, forms, controls, and application logic. This section provides an overview of Windows forms using Developer Studio 2006 for Win32 and outlines the steps you would use to build a simple Windows project.

In This Section

Windows Overview

This topic provides an overview of the different types of Windows applications and Windows components.

Building a Windows Application

This procedure describes the essential tasks to create a Windows application.

Windows Overview

The Windows platform provides several ways to help you create and build applications. The most common types of Windows applications are:

- GUI Applications
- Console Applications
- Service Applications
- Packages and DLLs

GUI Applications

A graphical user interface (GUI) application is designed using graphical components such as windows, menus, dialog boxes, and other features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created from your source files. The executable usually provides the basic functionality of your program. Simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable.

TheDeveloper Studio 2006 IDE offers two application UI models:

- Single Document Interface (SDI)
- Multiple Document Interface (MDI)

Single Document Interface

A SDI application normally contains a single document view.

Multiple Document Interface

In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors.

MDI applications require more planning and are more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. For instance, you need to set the FormStyle property of the TForm object to specify whether a form is a child (fsMDIChild) or main form (fsMDIForm). It is a best practice to define a base class for your child forms and derive each child form from this class. Otherwise, you will have to reset the form properties of the child. MDI applications often include a Window pop-up on the main menu that has items such as Cascade and Tile for viewing multiple windows in various styles. When a child window is minimized, its icon is located in the MDI parent form.

Console Applications

Console applications are 32-bit programs that run in a console window without a graphical interface. These applications typically do not require much user input and perform a limited set of functions. Any application that contains `{$APPTYPE CONSOLE}` in the code opens a console window of its own.

Service Applications

Service applications take requests from client applications, process those requests, and return the information to the client applications. Service applications typically run in the background without much user input. A Web, FTP, or an email server is an example of a service application.

Creating Packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs.

Packages are special DLLs used by Delphi applications, the IDE, or both. The two types of packages are runtime and designtime. Runtime packages provide functionality to a program while that program is running. Designtime packages extend the functionality of the IDE.

For most applications, packages provide greater flexibility and are easier to create than DLLs. However,here are a few situations where DLLs would work better than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a Web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass Delphi runtime type information (RTTI) across DLLs or from a DLL to an executable. If you pass an object from one DLL to another DLL or to an executable, you will not be able to use the is or as operators with the passed object. This is because the is and as operators need to compare RTTI. If you need to pass objects from a library, use packages instead of DLLs, because packages can share RTTI. Similarly, you should use packages instead of DLLs in Web Services because they rely on Delphi RTTI.

Building Web Applications with WebSnap

This section provides a conceptual background for building WebSnap applications using Developer Studio 2006. WebSnap makes it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers.

In This Section

Win32 Web Applications Overview
An overview of Win32 web applications programming using Web Snap and Web Broker.

Building a WebSnap Application
Describes the essential tasks to create an WebSnap application using Developer Studio 2006.

Win32 Web Applications Overview

This section covers:

- Web Application Support
- Web Broker Overview
- Web Snap Overview
- Debugging With the Web Application Debugger

For more detailed information on web applications, please see the Win32 Developers Guide in the Reference section of this Help system.

Win32 Web Application Support

The following types of web applications will be supported in Developer Studio 2006.

- ISAPI
- CGI
- Web Application Debugger

Apache web applications are not supported for this release.

ISAPI

Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file, and the required entries to the `uses` list and `exports` clause of the project file.

CGI

Selecting this type of application sets up your project as a console application, and adds the required entries to the `uses` clause of the project file.

Web Application Debugger

Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Web Broker Overview

Web Broker components, located on the **Internet** tab of the **Component Palette**, enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can construct HTML or XML documents within your program and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle multiple simultaneous, thread-safe, database connections.

Web Snap Overview

WebSnap augments Web Broker with additional components, wizards, and views, making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for

server-side scripting makes development and maintenance easier for teams of developers and Web designers. WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance.

The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a text editor. Changes can be made to the templates as needed, even after the application is deployed. There is no need to modify the project source code at all, which saves valuable development time. WebSnap's multiple module support can be used to divide your application into smaller pieces during the coding phases of your project, so that developers can work more independently.

Debugging With the Web Application Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server. The first time you run your application, it registers your COM server so that the Web Application Debugger can access it. Before you can run the Web Application Debugger, you will need to run `bin\serverinfo.exe` once to register the ServerInfo application.

Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it using the Web Application Debugger. You can set breakpoints in it just like any other executable. When you run your application, it displays the console window of the COM server that is your Web server application. Once you start your application and run the Web App Debugger, the ServerInfo page is displayed in your default browser, and you can select your application from a drop-down list. Once you have selected your application, click the **Go** button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Converting your application to another type of Web server application after debugging

When you have finished debugging your Web server application with the Web Application Debugger, you will need to convert it to another type that can be installed on a commercial Web server.

Building Web Services with Win32 Applications

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications. This section gives an overview of web services and web services support.

In This Section

Web Services Overview

This topic gives an overview of web services.

Using Web Services

Web Services Overview

Web Service applications are server implementations that do not require clients to use a specific platform or programming language. These applications define interfaces in a language-neutral document, and they allow multiple communication mechanisms.

Web Services are designed to work using Simple Object Access Protocol (SOAP). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. SOAP uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol.

Web Service applications use a Web Service Definition Language (WSDL) document to publish information on interfaces that are available and how to call them. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code when you import the WSDL document.

Building Database Applications for the Win32 Platform

Database applications let users interact with the information that is stored in the databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

In This Section

dbGo Overview
This topic describes the dbGo components in the Tool Palette.

dbExpress Components
This topic gives an overview of the dbExpress components in the Tool Palette.

BDE Overview
This topic gives an overview of the BDE components in the Tool Palette.

Deploying Multi-tiered Database Applications (DataSnap)

dbGo Overview

dbGo provides the developers with a powerful and logical object model for programmatically accessing, editing, and updating data from a wide variety of data sources through OLE DB system interfaces. The most common usage of dbGo is to query a table or tables in a relational database, retrieve and display the results in an application, and perhaps allow users to make and save changes to the data.

The ADO layer of an ADO-based application consists of the latest version of Microsoft ADO, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The **dbGo** category of the **Tool Palette** hosts the **dbGo** components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. The components require the latest version of ADO to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most **dbGo** components have direct counterparts in the components available for other data access mechanisms: a database connection component, **TADOConnection**, and various types of datasets. In addition, **dbGo** includes **TADOCommand**, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The main **dbGo** components are:

Components	Function
TADOConnection	A database connection component that establishes a connection with an ADO data store. Multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TRDSConnection	A database connection component to marshal data in multi-tier database applications that are built using ADO-based application servers.
TADODataSet	Primary dataset used for retrieving and operating on data. **TADODataSet** can retrieve data from a single or multiple tables, can connect directly to a data store, or use a **TADOConnection** component
TADOTable	A table-type dataset for retrieving and operating on a recordset produced by a single database table. **TADOTable** can connect directly to a data store or use a **TADOConnection** component
TADOQuery	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement. **TADOQuery** can also execute Data Definition Language (DDL) SQL statements. It can connect directly to a data store or use a **TADOConnection** component.
TADOStoredProc	A stored procedure-type dataset for executing stored procedures. **TADOStoredProc** executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a **TADOConnection** component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets). **TADOCommand** can be used with a supporting dataset component, or retrieve a dataset from a table. It can connect directly to a data store or use a **TADOConnection** component

dbExpress Components

dbExpress is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, dbExpress provides a driver that adapts the server-specific software to a set of uniform dbExpress interfaces. When you deploy a database application that uses dbExpress, you include a DLL(the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets.

dbExpress connections, tables, views, and stored procedures that show up in a data tree view support drag & drop with native and managed vcl forms.

The **dbExpress** category of the **Tool Palette** contains components that use **dbExpress** to access database information. They are:

Components	Function
TSQLConnection	Encapsulates a dbExpress connection to a database server
TSQLDataSet	Represents any data available through **dbExpress**, or sends commands to a database accessed through **dbExpress**
TSQLQuery	A query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any
TSQLTable	A table-type dataset that represents all of the rows and columns of a single database table
TSQLStoredProc	A stored procedure-type dataset that executes a stored procedure defined on a database server
TSQLMonitor	Intercepts messages that pass between an SQL connection component and a database server and saves them in a string list
TSimpleDataSet	A client dataset that uses an internal **TSQLDataSet** and **TDataSetProvider** for fetching data and applying updates

BDE Overview

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. The components on the **BDE** category of the **Tool Palette** enable you to connect to database information using the BDE.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broader range of support for database manipulation. Although you can use the API of the BDE directly in your application, the components on the **BDE** category of the **Tool Palette** wrap most of this functionality for you.

The main **BDE** components are:

Components	Function
TTable	Retrieves data from a physical database table via the BDE and supplies it to one or more data-aware components through a DataSource component. Conversely, it also sends data received from a component to a physical database via the BDE.
TQuery	Uses SQL statements to retrieve data from a physical database table via the BDE and supplies it to one or more data-aware components through a TDataSource component. Conversely, it uses SQL statements to send data from a component to a physical database via the BDE.
TStoredProc	Enables an application to access server stored procedures. It sends data received from a component to a physical database via the BDE.
TDatabase	Sets up a persistent connection to a database, especially a remote database requiring a user login and password.
TSession	Provides global control over a group of database components. A default TSession component is automatically created for each database application. You must use the TSession component only if you are creating a multithreaded database application. Each database thread requires its own session component.
TBatchMove	Copies a table structure or its data. It can be used to move entire tables from one database format to another.
TUpdateSQL	Lets you use cached updates support with read-only datasets.
TNestedTable	Retrieves the data in a nested dataset field and supplies it to data-aware controls through a datasource component.

Getting Started with InterBase Express

InterBase Express (IBX) is a set of data access components that provide a means of accessing data from InterBase databases. The InterBase Administration Components, which require InterBase 6, are described after the InterBase data access components.

IBX components

The following components are located on the InterBase tab of the component palette.

	TIBTable
	TIBQuery
	TIBStoredProc
	TIBDatabase
	TIBTransaction
	TIBUpdateSQL
	TIBDataSet
	TIBSQL
	TIBDatabaseInfo
	IBSQLMonitor
	TIBEvents
	TIBExtract
	TIBCustomDataSet

Though they are similar to BDE components in name, the IBX components are somewhat different. For each component with a BDE counterpart, the sections below give a discussion of these differences.

There is no simple migration from BDE to IBX applications. Generally, you must replace BDE components with the comparable IBX components, and then recompile your applications. However, the speed you gain, along with the access you get to the powerful InterBase features make migration well worth your time.

IBDatabase

Use a TIBDatabase component to establish connections to databases, which can involve one or more concurrent transactions. Unlike BDE, IBX has a separate transaction component, which allows you to separate transactions and database connections.

To set up a database connection:

1. Drop an IBDatabase component onto a form or data module.
2. Fill out the DatabaseName property. For a local connection, this is the drive, path, and filename of the database file. Set the Connected property to true.
3. Enter a valid username and password and click OK to establish the database connection.

Warning: Tip: You can store the username and password in the IBDatabase component's Params property by setting the LoginPrompt property to false after logging in. For example, after logging in as the system administrator and setting the LoginPrompt property to false, you may see the following when editing the Params property:

```
user_name=sysdba
password=masterkey
```

IBTransaction

Unlike the Borland Database Engine, IBX controls transactions with a separate component, TIBTransaction. This powerful feature allows you to separate transactions and database connections, so you can take advantage of the InterBase two-phase commit functionality (transactions that span multiple connections) and multiple concurrent transactions using the same connection.

Use an IBTransaction component to handle transaction contexts, which might involve one or more database connections. In most cases, a simple one database/one transaction model will do.

To set up a transaction:

1. Set up an IBDatabase connection as described above.
2. Drop an IBTransaction component onto the form or data module
3. Set the DefaultDatabase property to the name of your IBDatabase component.
4. Set the Active property to true to start the transaction.

IBX dataset components

There are a variety of dataset components from which to choose with IBX, each having their own characteristics and task suitability:

IBTable

Use an TIBTable component to set up a live dataset on a table or view without having to enter any SQL statements.

IBTable components are easy to configure:

1. Add an IBTable component to your form or data module.
2. Specify the associated database and transaction components.
3. Specify the name of the relation from the TableName drop-down list.
4. Set the Active property to true.

IBQuery

Use an TIBQuery component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables.

IBQuery components provide a read-only dataset, and adapt well to the InterBase client/server environment. To set up an IBQuery component:

1. Set up an IBDatabase connection as described above.
2. Set up an IBTransaction connection as described above.
3. Add an IBQuery component to your form or data module.
4. Specify the associated database and transaction components.
5. Enter a valid SQL statement for the IBQuery's SQL property in the String list editor.
6. Set the Active property to true

IBDataSet

Use an TIBDataSet component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables. IBDataSet components are similar to IBQuery components, except that they support live datasets without the need of an IBUpdateSQL component.

The following is an example that provides a live dataset for the COUNTRY table in employee.gdb:

1. Set up an IBDatabase connection as described above.
2. Specify the associated database and transaction components.
3. Add an IBDataSet component to your form or data module.
4. Enter SQL statements for the following properties:

SelectSQL	`SELECT Country, Currency FROM Country`
RefreshSQL	`SELECT Country, Currency FROM Country WHERE Country = :Country`
ModifySQL	`UPDATE Country SET Country = :Country, Currency = :Currency WHERE Country = :Old_Country`
DeleteSQL	`DELETE FROM Country WHERE Country = :Old_Country`
InsertSQL	`INSERT INTO Country (Country, Currency) VALUES (:Country, :Currency)`

1. Set the Active property to true.

Note: **Note:** Parameters and fields passed to functions are case-sensitive in dialect 3. For example,

```
FieldByName(EmpNo)
```

would return nothing in dialect 3 if the field was 'EMPNO'.

IBStoredProc

Use TIBStoredProc for InterBase executable procedures: procedures that return, at most, one row of information. For stored procedures that return more than one row of data, or "Select" procedures, use either IBQuery or IBDataSet components.

IBSQL

Use an TIBSQL component for data operations that need to be fast and lightweight. Operations such as data definition and pumping data from one database to another are suitable for IBSQL components.

In the following example, an IBSQL component is used to return the next value from a generator:

1. Set up an IBDatabase connection as described above.
2. Put an IBSQL component on the form or data module and set its Database property to the name of the database.
3. Add an SQL statement to the SQL property string list editor, for example:

```
SELECT GEN_ID(MyGenerator, 1) FROM RDB$DATABASE
```

IBUpdateSQL

Use an TIBUpdateSQL component to update read-only datasets. You can update IBQuery output with an IBUpdateSQL component:

1. Set up an IBQuery component as described above.
2. Add an IBUpdateSQL component to your form or data module.
3. Enter SQL statements for the following properties: DeleteSQL, InsertSQL, ModifySQL, and RefreshSQL.
4. Set the IBQuery component's UpdateObject property to the name of the IBUpdateSQL component.
5. Set the IBQuery component's Active property to true.

IBSQLMonitor

Use an TIBSQLMonitor component to develop diagnostic tools to monitor the communications between your application and the InterBase server. When the TraceFlags properties of an IBDatabase component are turned on, active IBSQLMonitor components can keep track of the connection's activity and send the output to a file or control.

A good example would be to create a separate application that has an IBSQLMonitor component and a Memo control. Run this secondary application, and on the primary application, activate the TraceFlags of the IBDatabase component. Interact with the primary application, and watch the second's memo control fill with data.

IBDatabaseInfo

Use an TIBDatabaseInfo component to retrieve information about a particular database, such as the sweep interval, ODS version, and the user names of those currently attached to this database.

For example, to set up an IBDatabaseInfo component that displays the users currently connected to the database:

1. Set up an IBDatabase connection as described above.
2. Put an IBDatabaseInfo component on the form or data module and set its Database property to the name of the database.
3. Put a Memo component on the form.
4. Put a Timer component on the form and set its interval.
5. Double click on the Timer's OnTimer event field and enter code similar to the following:

```
Memo1.Text := IBDatabaseInfo.UserNames.Text;   //  Delphi example
    Memo1->Text = IBDatabaseInfo->UserNames->Text; //  C++ example
```

IBEvents

Use an *IBEvents* component to register interest in, and asynchronously handle, events posted by an InterBase server.

To set up an IBEvents component:

1. Set up an IBDatabase connection as described above.
2. Put an IBEvents component on the form or data module and set its Database property to the name of the database.
3. Enter events in the Events property string list editor, for example:

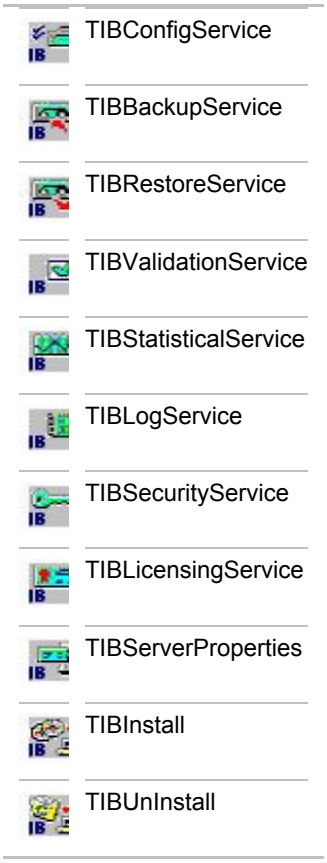
```
IBEvents.Events.Add('EVENT_NAME');   //  Delphi example
    IBEvents->Events->Add("EVENT_NAME"); //  C++ Example
```

1. 4. Set the Registered property to true.

InterBase Administration Components

If you have InterBase 6 installed, you can use the InterBase 6 Administration components, which allow you to use access the powerful InterBase Services API calls.

The components are located on the InterBase Admin tab of the IDE and include:



Note: You must install InterBase 6 to use these features.

IBConfigService

Use an TIBConfigService object to configure database parameters, including page buffers, async mode, reserve space, and sweep interval.

IBBackupService

Use an TIBBackupService object to back up your database. With IBBackupService, you can set such parameters as the blocking factor, backup file name, and database backup options.

IBRestoreService

Use an TIBRestoreService object to restore your database. With IBRestoreService, you can set such options as page buffers, page size, and database restore options.

IBValidationService

Use an TIBValidationService object to validate your database and reconcile your database transactions. With the IBValidationService, you can set the default transaction action, return limbo transaction information, and set other database validation options.

IBStatisticalService

Use an TIBStatisticalService object to view database statistics, such as data pages, database log, header pages, index pages, and system relations.

IBLogService

Use an TIBLogService object to create a log file.

IBSecurityService

Use an TIBSecurityService object to manage user access to the InterBase server. With the IBSecurityService, you can create, delete, and modify user accounts, display all users, and set up work groups using SQL roles.

IBLicensingService

Use an TIBLicensingService component to add or remove InterBase software activation certificates.

IBServerProperties

Use an TIBServerProperties component to return database server information, including configuration parameters, and version and license information.

IBInstall

Use an TIBInstall component to set up an InterBase installation component, including the installation source and destination directories, and the components to be installed.

IBUnInstall

Use an TIBUnInstall component to set up an uninstall component.

Building Applications with VCL Components

VCL is a set of visual components for the rapid development of Windows applications in the Delphi language. VCL contains a wide variety of visual, non-visual, and utility classes for tasks such as building Windows applications, web applications, database applications, and console applications.

In This Section

VCL Overview
An overview of using the VCL for application development.

Building a VCL Forms Application
Describes the essential tasks to create a VCL Forms application using Developer Studio 2006.

VCL Overview

This section introduces:

- VCL Architecture
- VCL versus VCL.NET
- VCL Components
- Working With Components

VCL Architecture

VCL is an acronym for the Visual Component Library, a set of visual components for rapid development of Windows applications in the Delphi language. VCL contains a wide variety of visual, non-visual, and utility classes for tasks such as Windows application building, web applications, database applications, and console applications. All classes descend from TObject. TObject introduces methods that implement fundamental behavior like construction, destruction, and message handling.

VCL versus VCL.NET

VCL.Net contains only a subset of the full functionality available in VCL for Win32. The .NET Framework was architected to accommodate any .NET-compliant language. In many cases, Delphi source code that operates on Win32 VCL classes and functions recompiles with minimal changes on .NET. In some cases, the code recompiles with no changes at all. Since VCL.NET is a large subset of VCL, it supports many of the existing VCL classes. However, source code that calls directly to the Win32 API requires source code changes.

VCL Components

Components are a subset of the component library that descend from the class TComponent. You can place components on a form or data module and manipulate them at designtime. Using the **Object Inspector**, you can assign property values without writing code. Most components are either visual or nonvisual, depending on whether they are visible at runtime. Some components appear on the **Component Palette**.

Visual Components

Visual components, such as TForm and TSpeedButton, are called controls and descend from TControl. Controls are used in GUI applications, and appear to the user at runtime. TControl provides properties that specify the visual attributes of controls, such as their height and width.

NonVisual Components

Nonvisual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a TDataSource component on a form to connect a control and a dataset used by the control. This connection is not visible to the user, so TDataSource is nonvisual. At designtime, nonvisual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control.

Other VCL Classes

Classes that are not components (that is, classes that descend from TObject but not TComponent) are also used for a variety of tasks. Typically, these classes are used for accessing system objects (such as a file or the clipboard)

or for transient tasks (such as storing data in a list). You cannot create instances of these classes at designtime, although they are sometimes created by the components that you add in the **Form Designer**.

Working With Components

Many components are provided in the IDE on the **Component Palette**. You select components from the **Component Palette** and place them onto a form or data module. You design the user interface of an application by arranging the visual components such as buttons and list boxes on a form. You can also place nonvisual components, such as data access components, on either a form or a data module. At first, Delphi's components appear to be just like any other classes. But there are differences between components in Delphi and the standard class hierarchies that many programmers work with. Some differences are:

- All Delphi components descend from TComponent.
- Components are most often used as is. They are changed through their properties, rather than serving as base classes to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- Components can only be allocated on the heap, not on the stack.
- Properties of components contain runtime type information.
- Components can be added to the **Component Palette** in the IDE and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider a dialog box containing a button. In a Windows program developed using VCL components, when a user clicks the button, the system generates a `WM_LBUTTONDOWN` message. The program must catch this message (typically in a switch statement, a message map, or a response table) and send it to a routine that will execute in response to the message. Most Windows messages (VCL applications) are handled by Delphi components. When you want to respond to a message or system event, you only need to provide an event handler.

Using Events

Almost all the code you write is executed, directly or indirectly, in response to events. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event, called an event handler, is a Delphi procedure.

The **Events** page of the **Object Inspector** displays all events defined for a given component. Double-clicking an event in the **Object Inspector** generates a skeleton event handling procedure, which you can fill in with code to respond to that event. Not all components have events defined for them.

Some components have a default event, which is the event the component most commonly needs to handle. For example, the default event for a button is `OnClick`. Double-clicking on a component with a default event in the **Form Designer** will generate a skeleton event handling procedure for the default event.

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop down menu commands. When a button performs the same action as a menu command, you can write a single event handler and then assign it to the OnClick event for both the button and the menu item by setting the event handler in the **Object Inspector** for both the events you want to respond to.

This is the simplest way to reuse event handlers. However, action lists, and in the VCL, action bands, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications; action bands cannot.

Setting Component Properties

To set published properties at design time, you can use the **Object Inspector** and, in some cases, property editors. To set properties at runtime, assign their values in your application source code.

When you select a component on a form at design time, the **Object Inspector** displays its published properties and, when appropriate, allows you to edit them.

When more than one component is selected, the **Object Inspector** displays all properties—except `Name`—that are shared by the selected components. If the value for a shared property differs among the selected components, the **Object Inspector** displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Changing code-related properties, such as the name of an event handler, in the **Object Inspector** automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, are immediately reflected in the **Object Inspector**.

Building Interoperable Applications

Developer Studio 2006 provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

In This Section

Building COM Applications

Describes the chapter content in a sentence; used for part descriptions only.

Interoperable Applications Procedures

Building COM Applications

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients and servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

This topic covers:

- COM Technologies Overview
- COM Interfaces
- COM Servers
- COM Clients

COM Technologies Overview

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The most important aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects conform to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions for creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly (such as Active Documents). You can find these wrappers defined in the `ComObj` unit and the API definitions in the `AxCtrls` unit.

Note: Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the `AxCtrls`, `OleCtrls`, and `OleServer` units. In addition, the Delphi interface to the COM API is in `ActiveX.pas` and `ComSvcs.pas`.

COM Interfaces

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients.

For example, every COM object must implement the basic interface, `IUnknown`. Through a routine called `QueryInterface` in `IUnknown`, clients can request other interfaces implemented by the server.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to tell the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as `IMalloc` or `IPersist`.
- Interfaces are guaranteed to have a unique identification, called a Globally Unique Identifier (GUID), which is a 128-bit randomly generated number. Interface GUIDs are called Interface Identifiers (IIDs). This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer, either explicitly or implicitly.
- Interfaces are not objects themselves, they provide a way to access an object. Therefore, clients do not access data directly, they access data through an interface pointer. Windows 2000 adds another layer of indirection, known as an interceptor, through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the base interface, `IUnknown`.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection.

The IUnknown interface

All COM objects must support the fundamental interface, called `IUnknown`, a typedef to the base interface type `IInterface`. `IUnknown` contains the following routines:

- `QueryInterface`: Provides pointers to other interfaces that the object supports.
- `AddRef` and `Release`: Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the `IUnknown` method, `QueryInterface`. `QueryInterface` knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the `IUnknown` methods, `AddRef` and `Release`, which are simple reference counting methods. As long as the reference count of an object is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object.

COM Interface Pointers

An interface pointer is a pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a vtable. Vtables are similar to the mechanism used to support virtual functions in Delphi. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Delphi classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer to the pointer to the vtable.

In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, another level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-

in-time activation, where the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

COM Servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know how a COM object performs its service; the object's implementation remains hidden. An object makes its services available through its interfaces as described previously.

In addition, clients do not need to know where a COM object resides. COM provides transparent access regardless of the object's location.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server create an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).
- Implement a class factory object, which creates another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

COM Clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are Automation controllers, which control an Automation server and ActiveX containers, which host an ActiveX control.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or objects that appear in the container's user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

The task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server object and uses its properties and methods. Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. Delphi lets you wrap the server `CoClass` in a component on the client, which you can even install on the Component palette. Samples of such component wrappers appear on two pages of the Component palette, sample ActiveX wrappers appear on the ActiveX page, and sample Automation objects appear on the Servers page.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information.

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the `IDispatch` interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations

about the client using its objects. Similarly, clients don't need to know how an object provides the services, they simply rely on server objects to provide the services they describe in their interfaces.

COM Extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories. In addition, when working in a large, distributed environment, you can create transactional COM objects. Prior to Windows 2000, these objects were not an architectural part of COM, but ran in the Microsoft Transaction Server (MTS) environment. As of Windows 2000, this support is integrated into COM+. Delphi provides wizards to easily implement applications that use the above technologies in the Delphi environment.

Automation Servers

Automation refers to the ability of an application to control the objects in another application programmatically, such as a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller. Automation can be used on in-process, local, and remote servers.

Automation is defined by two major points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and the arguments to those methods. Typically, this information is available in a type library. The Automation server can also generate type information dynamically when queried via its `IDispatch` interface.
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the `IDispatch` interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space, because the Automation `IDispatch` interface automates the marshaling process. Automation does, however, restrict the types that you can use.

Active X Controls

Delphi wizards allow you to easily create ActiveX controls. ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications, which need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as already written OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, TOleContainer, to support linking and embedding of existing Active Documents.

You can also use TOleContainer as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note: While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

Transactional Objects

Delphi uses the term "transactional objects" to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment.

The transaction services provide robustness so that activities are always either completed or rolled back. The server never partially completes an activity. The security services allow you to expose different levels of support to different classes of clients. The resource management allows an object to handle more clients by pooling resources or keeping objects active only when they are in use. To enable the system to provide these services, the object must implement the `IObjectControl` interface. To access the services, transactional objects use an interface called `IObjectContext`, which is created for them by MTS or COM+.

Under MTS, the server object must be built into a DLL library, which is then installed in the MTS runtime environment. That is, the server object is an in-process server that runs in the MTS runtime process space. Under COM+, this restriction does not apply because all COM calls are routed through an interceptor. To clients, the difference between MTS and COM+ is transparent.

MTS or COM+ servers group transactional objects that run in the same process space. Under MTS, this group is called an MTS package, while under COM+ it is called a COM+ application. A single machine can be running several different MTS packages (or COM+ applications), where each one is running in a separate process space.

To clients, the transactional object may appear like any other COM server object. The client does not need know about transactions, security, or just-in-time activation unless it is initiating a transaction itself.

Both MTS and COM+ provide a separate tool for administering transactional objects. This tool lets you configure objects into packages or COM+ applications, view the packages or COM+ applications installed on a computer, view or change the attributes of the included objects, monitor and manage transactions, make objects available to clients, and so on. Under MTS, this tool is the MTS Explorer. Under COM+ it is the COM+ Component Manager.

Type Libraries

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their

interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor.

Build Configurations

This section contains topics about build configurations. Build configurations store sets of command-line options for build tools such as the compiler and linker.

Build configurations are available for only the C++ personality.

In This Section

Managing C++ Build Configurations
Describes the behavior of build configurations in the C++ personality.

Managing C++ Build Configurations

A build configuration stores sets of command-line options for build tools such as the compiler and linker. You can create multiple build configurations that you can quickly switch between.

Two configurations are available by default: **Debug Build** and **Release Build**. Use the **Build Configurations** dialog box, accessable by choosing **Project** ▸ **Options** and clicking **Configurations**, to create other build configurations.

Through the features available in the C++ **Project Options** dialog box and the way in which build configurations inherit from a top-level configuration, you can easily manage multiple sets of command-line options.

Build Configuration Inheritance

Two configurations are available by default: **Debug Build** and **Release Build**. These build configurations and any others you create inherit their options from the top-level **All Configurations** build configuration. When you change options in **All Configurations**, the changes affect all other available configurations.

For example, if you had several build configurations and wanted all of them to generate a Map file with publics, you would select **All Configurations** and enable that option. All other build configurations would then inherit the **Generate Map file with publics** option.

The Inherit Check Box

The **Inherit** check box appears next to an option if

- The option requires a list of strings as a parameter, such as the bcc32 -D**Conditional Defines** option.
- You are currently editing a build configuration other than **All Configurations**.

When you check **Inherit**, the current build configuration inherits the strings defined in **All Configurations**. You cannot delete any inherited strings from the current build configuration.

Overriding Inherited Options

Build configurations inherit options from the **All Configurations** build configuration. When the value of an option differs from the value in **All Configurations**, the option appears in blue and has overridden the inherited option. You can right-click an overridden option to revert its value.

Setting Project Defaults

If you check **Default** on the **Project Options** dialog box, the current project settings will be used for projects you create later. The current set of build configurations is also saved.

For example, if you created a new build configuration called **Internal Development Build** with **Default** checked in the **Project Options** dialog box, the **Internal Development Build** will be available for projects you create later.

Tip: To restore the factory default project settings, delete the default project file located in C:\Documents and Settings\<user-name>\local settings\application data\Borland\BDS\4.0\DefProject.bdsproj

Debugging C++ Applications with CodeGuard Error Reporting

CodeGuard provides runtime debugging for C++ applications developed with Developer Studio 2006. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.

In This Section

CodeGuard Overview
CodeGuard Errors
CodeGuard Warnings

CodeGuard Overview

CodeGuard provides runtime debugging for C++ applications developed with Developer Studio 2006. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.

CodeGuard provides two principal types of coverage:

- Memory and Resource Use
- Function Call Validation

Memory and Resource Use

CodeGuard checks for faulty memory use, improper memory allocation or deallocation, invalid file streams or handles, and resource leaks caused by improper use of file streams or handles. CodeGuard verifies pointer dereferencing and pointer arithmetic. CodeGuard can report an error if your program tries to access memory or resources that have already been released.

Function Call Validation

CodeGuard verifies function arguments and reports function failure as indicated by the return value of the function. It validates Windows resource handles used in function calls.

CodeGuard Errors

CodeGuard reports four types of runtime errors.

In This Section

Access Errors
Resource Errors
Exception Errors
Function Failure Errors

Access Errors

Access errors result from improper memory management.

When CodeGuard detects accesses to freed memory blocks or deleted objects, it can identify where each block was allocated and deleted. Enable the **Delay Free** option using the **CodeGuard Configuration** dialog box to use this feature.

The following are types of access errors:

- Access in freed memory
- Access in uninitialized stack
- Access in invalid stack

Access In Freed Memory

In the following example, CodeGuard identifies the line where an invalid access occurrs. CodeGuard then indicates where the memory block was allocated and subsequently freed.

```
Error 00004. 0x100430 (Thread 0xFFF87283):
Access in freed memory: Attempt to access 19 byte(s) at 0x00B423DC.
strcpy(0x00B423DC, 0x004091CA ["Copy to free block"])
| lang.cpp line 106:
|
|    free(buf_h);
|>   strcpy(buf_h, "Copy to free block");
|
|    //----------------------//
Call Tree:
   0x004011F1(=LANG.EXE:0x01:0001F1) lang.cpp#106
   0x00407EE5(=LANG.EXE:0x01:006EE5)

The memory block (0x00B423DC) [size: 21 bytes] was allocated with malloc
| lang.cpp line 80:
|    char * pad = (char *) malloc(200);
|    // An array in the RTL heap.
|>   char * buf_h = (char *) malloc(21);
|    char * p;
|    // A scratch buffer.
Call Tree:
   0x004011A1(=LANG.EXE:0x01:0001A1) lang.cpp#80
   0x00407EE5(=LANG.EXE:0x01:006EE5)

The memory block (0x00B423DC) was freed with free
| lang.cpp line 105:
```

```
|    //-------------//
|
|>   free(buf_h);
|    strcpy(buf_h, "Copy to free block");
|
Call Tree:
   0x004011E5(=LANG.EXE:0x01:0001E5) lang.cpp#105
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Access In Uninitialized Stack

In the following example, the pointer `p` became invalid when `getBadLocal` returned from execution. No additional information is provided because the stack frame for `getBadLocal` was automatically removed.

```
Error 00005. 0x120400 (Thread 0xFFF87283):
Access in uninitialized stack: Attempt to access 20 byte(s) at 0x0072FC88.
memcpy(0x0072FCC4, 0x0072FC88, 0x14 [20])
| lang.cpp line 112:
|    //----------------------//
|    p = getBadLocal();
|>   memcpy(buffer, p, 20);
|
|    //-------------//
Call Tree:
   0x00401208(=LANG.EXE:0x01:000208) lang.cpp#112
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Access In Invalid Stack

In the following example, an allocation was made for `buf_s` on the stack. However, the `strcpy` function writes just below the beginning of the valid stack region. CodeGuard identifies this as an error even if the string is only one byte long.

```
Error 00002. 0x110400 (Thread 0xFFF87283):
Access in invalid stack: Attempt to access 22 byte(s) at 0x0072FD8F.
strcpy(0x0072FD8F, 0x00409188 ["This string is long!\n"])
| LANG.CPP line 93:
|
|    // Stack underrun:
|>   strcpy(buf_s -1, "This string is long!\n");
|
|    // Global data overrun:
Call Tree:
   0x004011C5(=LANG.EXE:0x01:0001C5) LANG.CPP#93
   0x00407EED(=LANG.EXE:0x01:006EED)
```

Resource Errors

Resources are memory blocks (allocated with functions like `malloc`, `GlobalAlloc`) and object arrays, such as file handles, stream handles, modules, and items returned by new[].

The following runtime error examples illustrate how CodeGuard reports improper use of resources:

- Bad parameter

- Reference to freed resource
- Resource type mismatch
- Resource leaks
- Resource from different RTL

Bad Parameter

When a resource is passed to a function, CodeGuard checks the runtime arguments. CodeGuard notifies you if it detects a bad parameter.

```
Error 00017. 0x310000 (Thread 0xFFF87283):
Bad parameter: A bad file handle (0xEA) has been passed to the function.
close(0xEA [234])
| lang.cpp line 170:
|    // using a bad handle //
|    //--------------------//
|>   close(234);
|
|    //----------------------//
Call Tree:
   0x00401456(=LANG.EXE:0x01:000456) lang.cpp#170
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Reference To Freed Resource

In the following example, CodeGuard reports an attempt to read from a file that has already been closed. The CodeGuard log shows where the file was opened and subsequently closed.

```
Error 00020. 0x310030 (Thread 0xFFF840F1):
Reference to freed resource:
read(0x3 [3], 0x0072FCC4, 0x5 [5])
| lang.cpp line 177:
|    int i = open("lang.cpp", 0);
|    close(i);
|>   read (i, buffer, 5);
|
|    //--------------//
Call Tree:
   0x00401487(=LANG.EXE:0x01:000487) lang.cpp#177
   0x00407EED(=LANG.EXE:0x01:006EED)

The file handle (0x00000003) [name: 'lang.cpp'] was opened with open
| lang.cpp line 175:
|    // using a freed handle //
|    //----------------------//
|>   int i = open("lang.cpp", 0);
|    close(i);
|    read (i, buffer, 5);
Call Tree:
   0x0040146C(=LANG.EXE:0x01:00046C) lang.cpp#175
   0x00407EED(=LANG.EXE:0x01:006EED)

The file handle (0x00000003) was closed with close
| lang.cpp line 176:
```

```
|    //----------------------//
|    int i = open("lang.cpp", 0);
|>   close(i);
|    read (i, buffer, 5);
|
Call Tree:
   0x00401477(=LANG.EXE:0x01:000477) lang.cpp#176
   0x00407EED(=LANG.EXE:0x01:006EED)
```

Resource Type Mismatch

In the following example, a memory block that was allocated with the new[] operator, and should therefore be released with the delete[] operator, is instead released with a call to the `free` function.

```
Error 00024. 0x350010 (Thread 0xFFF840F1):
Resource type mismatch: a(n) memory block was expected.
free(0x00B42464)
| lang.cpp line 188:
|    //---------------//
|    char * ss = new char[21];
|>   free(ss);
|
| #ifdef    __WIN32__
Call Tree:
   0x0040149F(=LANG.EXE:0x01:00049F) lang.cpp#188
   0x00407EED(=LANG.EXE:0x01:006EED)

The object array (0x00B42464) [size: 21 bytes] was created with new[]
| lang.cpp line 187:
|    // type mismatch //
|    //---------------//
|>   char * ss = new char[21];
|    free(ss);
|
Call Tree:
   0x00401498(=LANG.EXE:0x01:000498) lang.cpp#187
   0x00407EED(=LANG.EXE:0x01:006EED)
```

Resource Leaks

In the following example, memory has been allocated but is never freed.

```
The memory block (0x00B42310) [size: 200 bytes] was allocated with malloc
| lang.cpp line 78:
|    // An array on the stack.
|    char buf_s[21];
|>   char * pad = (char *) malloc(200);
|    // An array in the RTL heap.
|    char * buf_h = (char *) malloc(21);
Call Tree:
   0x00401199(=LANG.EXE:0x01:000199) lang.cpp#78
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Resource From Different RTL

CodeGuard reports an error if your application allocates, uses, or releases resources in different versions of the runtime library. This can happen, as the following example illustrates, if you link with a static runtime library but call a DLL.

Note: CodeGuard detects resource type mismatches before it detects mixed versions of the RTL. When the two kinds of error are combined, CodeGuard will not report the mixed RTLs until you correct the resource type mismatch.

```
Error 00001. 0x340010 (Thread 0x0062):
Resource from different RTL:
close(0x3 [3])
| testdll.cpp line 23:
| {¬
|    MessageBox(NULL,"RTLMixHandle: DLL closing EXE handle", "TESTDLL.CPP", MB_OK );
|>   close( handle );
|    return 1;
| }
Call Tree:
   0x0032115A(=testdll.dll:0x01:00015A) testdll.cpp#23
   0x00401660(=WINAPI.EXE:0x01:000660) filescg.cpp#33
   0x00401271(=WINAPI.EXE:0x01:000271) winapi.cpp#122
   0x77EA15B3
   0x00408B9A(=WINAPI.EXE:0x01:007B9A)

The file handle (0x00000003) [name: 'test2.dat'] was opened with open
| filescg.cpp line 32:
|
|  MessageBox(NULL,"FilesMixCG: Mixing RTL file handles", "FILESCG.CPP", MB_OK );
|> i = open("test2.dat", O_CREAT, S_IREAD | S_IWRITE );
|  RTLMixHandle( i );
| }
Call Tree:
   0x00401657(=WINAPI.EXE:0x01:000657) filescg.cpp#32
   0x00401271(=WINAPI.EXE:0x01:000271) winapi.cpp#122
   0x77EA15B3
   0x00408B9A(=WINAPI.EXE:0x01:007B9A)
```

Exception Errors

When a system exception occurs, CodeGuard reports the runtime error using information provided by the operating system. If possible, the CodeGuard log shows where your application caused the exception. CodeGuard does not trap or redirect the exception or otherwise interfere with normal program behavior.

The following exceptions illustrate how CodeGuard exception reporting:

- General Protection Fault
- Divide by zero

General Protection Fault

In the following example, CodeGuard provides information on a general protection fault (Intel system exception `0xD`). In addition to the location of the source code that caused the exception, the log shows where the memory was

allocated and subsequently freed. The reported incorrect value is a result of accessing a byte pattern that CodeGuard uses to identify invalid memory locations.

```
Error 00003. 0x400003 (Thread 0x0090):
Exception 0xC0000005: Access violation at 0x80828082.
| gpfault.c line 32:
|    {¬
|       q = p[3];
|>      *q = 1;
|    }
| }
Call Tree:
   0x004010E5(=GPFAULT.EXE:0x01:0000E5) gpfault.c#32
   0x00406B29(=GPFAULT.EXE:0x01:005B29)

The bogus value (0x80828082) was most likely retrieved by accessing a(n)
 memory block that has already been freed
The memory block (0x008322A4) [size: 16 bytes] was allocated with malloc
| gpfault.c line 17:
|     int *q;
|
|>   p = malloc(sizeof(*p) * 4);
|
|    /*  Initialize p */
Call Tree:
   0x00401094(=GPFAULT.EXE:0x01:000094) gpfault.c#17
   0x00406B29(=GPFAULT.EXE:0x01:005B29)

The memory block (0x008322A4) was freed with free
| gpfault.c line 17:
|     int *q;
|
|>   p = malloc(sizeof(*p) * 4);
|
|    /*  Initialize p */
Call Tree:
   0x00401094(=GPFAULT.EXE:0x01:000094) gpfault.c#17
   0x00406B29(=GPFAULT.EXE:0x01:005B29)
```

Divide By Zero

In the following example, CodeGuard identifies the location in source code where division by zero (Intel system exception `0x0`) occurred.

```
Error 00001. 0x400000 (Thread 0x008B):
Exception 0xC0000094:
| ZERODIV.C line 9:
| {¬
|  x = 1;
|> return x / y;
| }
|
Call Tree:
   0x0040109C(=ZERODIV.EXE:0x01:00009C) ZERODIV.C#9
   0x00406321(=ZERODIV.EXE:0x01:005321)
```

Function Failure Errors

CodeGuard reports function calls that fail, as indicated by their return value.

In the following example, the `close` function is given an invalid file handle, which causes it to return a value indicating that it was unable to close a file.

```
Error 00009. 0x820000 (r) (Thread 0xFFF840F1):
Function failure:
close(0x80868086 [-2138668922])=0xFFFFFFFF [-1]
| lang.cpp line 125:
|    // uninitialized data usage //
|    //--------------------------//
|>   close(m->handle);
|
|
Call Tree:
   0x00401236(=LANG.EXE:0x01:000236) lang.cpp#125
   0x00407EED(=LANG.EXE:0x01:006EED)
```

CodeGuard Warnings

CodeGuard can report situations where your application may access memory beyond a buffer's maximum size. Warnings are available for three types of runtime library functions.

In This Section

String Comparison Warnings
Memory Block Comparison Warnings
Pathname Merging and Splitting Warnings

String Comparison Warnings

Each of the following functions has a parameter that determines the maximum number of bytes it compares:

- `strncmp`
- `strnicmp`
- `strncmpi`
- `_fstrncmp`
- `_fstrnicmp`

If the **Warnings** option is enabled for the functions listed, CodeGuard verifies that a string comparison can be performed for each buffer passed to the function. If the buffer size is too large, as determined by the parameter passed to the function, and the buffer is not null-terminated, CodeGuard generates a warning.

If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the memory block is invalid, CodeGuard generates an error message.

Memory Block Comparison Warnings

Each of the following functions has a parameter that determines the maximum number of bytes it compares:

`memcmp`

`memicmp`

`_fmemcmp`

`_fmemicmp`

If the **Warnings** option is enabled for the functions listed above, CodeGuard verifies that a comparison can be performed for each memory block passed to the function. If a memory block is too large, as determined by the parameter passed to the function, CodeGuard generates a warning.

If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the memory block is invalid, CodeGuard generates an error message.

Pathname Merging and Splitting Warnings

Each of the following functions use constants defined in dir.h to determine the maximum number of bytes to copy to or from a buffer:

`fnmerge`

`fnsplit`

`getcurdir`

fnmerge

If the **Warnings** option is enabled, the output buffer is validated against `MAXPATH` before `fnmerge` is called.

If the **Warnings** option is disabled, the size of the output buffer is validated against the null-terminated string length after `fnmerge` is called.

fnsplit

If the **Warnings** option is enabled, the input buffers are validated against `MAXDRIVE`, `MAXDIR`, `MAXFILE`, and `MAXEXT` before `fnsplit` is called.

If the **Warnings** option is disabled, the input buffers are validated against the length of the null-terminated string after `fnsplit` is called.

getcurdir

If the **Warnings** option is enabled, the output buffer is validated against `MAXDIR` before `getcurdir` is called.

If the **Warnings** option is disabled, the output buffer is validated against the length of the null-terminated string after `getcurdir` is called.

Building Reports for Win32 Applications

Developer Studio 2006 ships with Rave Reports from Nevrona. Using the report components, you can build full-featured reports for your applications. You can create solutions that include reporting capabilities which can be used and customized by your customers. Additionally, the ComponentOne tools that ship with Developer Studio 2006 include components for creating and generating reports.

In This Section

Using Rave Reports in Developer Studio 2006
Describes how Developer Studio 2006 supports integration of Rave Reports objects.

Using Rave Reports in Developer Studio 2006

The Developer Studio 2006 environment supports the integration of report objects in your applications. This integration allows you to create a report using the Rave Reports Designer directly from within the Developer Studio 2006 IDE. Your application users can create and display their own reports, or display existing reports.

Creating New Reports in Developer Studio 2006

You can include reports in Developer Studio 2006 just as you would other 3rd-party components. The report is stored as a separate Rave Report object. You can reference the report in other applications that need to call or generate that report. When you create a new application, you can include the report object by adding a reference to it in the **Project Manager**. Rave Reports also provide the capability to connect your report object to a datasource, which allows your application to build the report dynamically, based on current database information.

Procedures

CodeGuard Procedures

Using CodeGuard

To run C++ application with CodeGuard reporting

1. Enable the CodeGuard reporting tool.
2. Enable CodeGuard compiler options for your project.
3. Choose **Run** ▸ **Run** to run your application.

During the execution of your application, CodeGuard runtime errors appear in the **Message** view.

CodeGuard also generates an error log named <project_name>.cgl that lists any errors it finds. The error log is located in the same directory as your executable.

Note: If you suspect that your program accesses a freed memory block but CodeGuard does not report an error, increase the value of **Maximum memory block size** or **Delay queue length** on the **Resource Options** page of the **Configure CodeGuard** dialog box.

To enable the CodeGuard reporting tool

1. Choose **Tools** ▸ **CodeGuard Configuration** to display the **CodeGuard Configuration** dialog box.
2. Verify that CodeGuard is enabled.
3. Click **OK**.

Note: If you change any CodeGuard settings in the **CodeGuard Configuration** dialog box, CodeGuard generates a .cgi configuration file with the same name and directory as your project file.

To enable CodeGuard compiler options for your project

1. Choose **Project** ▸ **Options** ▸ **C++ Compiler** ▸ **CodeGuard compile support** to display the CodeGuard compiler options.
2. Check **All CodeGuard options on** to enable full CodeGuard coverage.
3. Click **OK**.
4. Rebuild your project.

Note: If you compile and link your project in separate steps, remember to include the CodeGuard library (cg32.lib) before including other libraries.

Database Procedures

Accessing Schema Information

The schema information or metadata includes information about what tables and stored procedures are available on the server and the information about these tables and stored procedures (like the fields of a table, the indexes that are defined, and the parameters a stored procedure uses).

To access schema information

1. To populate a unidirectional dataset with metadata from the database server, call SetSchemaInfo method to indicate what data you want to see.
2. Set the type of schema information parameter of SetSchemaInfo method.
3. Set the name of table or stored procedure parameter of SetSchemaInfo method.
4. To fetch data after using the dataset for metadata, do one of the following:

- Set the CommandText property to specify the query, table, or stored procedure from which you want to fetch data.
- Set the type of schema information to stNoSchema and call SetSchemaInfo method.

Note: If you choose the second option, the dataset fetches the data specified by the CommandText property.

Configuring TSQL Connection

The first step when working with a unidirectional dataset is to connect it to a database server. At designtime, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server. The connection to a database server is represented by a separate **TSQLConnection** component. You work with **TSQLConnection** like any other database connection component.

To configure a TSQL Connection

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLConnection** component to the form.
4. Identify the driver.
5. Specify connection parameters.
6. Identify a database connection.
7. Display and use the **dbExpress Connection Editor**.

To identify the driver

1. Select the **TSQLConnection** component.
2. In the **Object Inspector**, set the DriverName property, to an installed dbExpress driver.
3. Identify the files associated with the driver name. Select any of the following:

   - The dbExpress driver
   - The dynamic link library

Note: The relationship between the dbExpress driver or dynamic link library and the database name is stored in a file called dbxdrivers.ini, which is updated when you install a dbExpress driver. The SQL connection component looks the dbExpress driver and the dynamic-link library up in dbxdrivers.ini when given the value of DriverName. When you set the DriverName property, **TSQLConnection** automatically sets the LibraryName and VendorLib properties to the names of the associated dlls. Once LibraryName and VendorLib have been set, your application does not need to rely on dbxdrivers.ini.

To specify a connection parameter

1. Double-click on the Params property in the **Object Inspector** to edit the parameters using **Value List Editor** at designtime.
2. Use the Params.Values property to assign values to individual parameters at run time.

To identify a database connection

1. Set the ConnectionName property to a valid connection name.

   This automatically sets the DriverName and Params properties.

2 Edit the Params property to change the saved set of parameter values.

3 Set the LoadParamsOnConnect property to **True** to develop your application using one database and deploy it using another.

   This causes **TSQLConnection** to automatically set DriverName and Params to the values associated with ConnectionName in dbxconnections.ini when the connection is opened.

4 Call the LoadParamsFromIniFile method.

   This method sets DriverName and Params to the values associated with ConnectionName in dbxconnections.ini (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

To display the Connection Editor

1 Double-click the **TSQLConnection** component.

   The **dbExpress Connection Editor** appears, with a drop-down drivers list, a list of connection names for the currently selected driver, and a connection parameters table for the currently selected connection name.

2 From the **Driver Name** drop-down list, select a **driver** to indicate the connection to use.

3 From the **Connection Name** list, select a **connection name**.

4 Choose the configuration that you want.

5 Click the **Test Connection** button to check for a valid configuration.

To define and modify connections using the Connection Editor

1 To edit the currently selected named connections in dbxconnections.ini, edit the parameter values in the parameter table.

2 Click **OK**.

   The new parameter values are saved to dbxconnections.ini.

3 Click the **Add Connection** button to define a new connection.

   The **New Connection** dialog appears.

4 In the **New Connection** dialog box, set the **Driver Name** and the **Connection Name**.

5 Click **OK**.

6 Click the **Delete Connection** button to delete the currently selected named connection from dbxconnections.ini.

7 Click the **Rename Connection** button to change the name of the currently selected named connection.

Connecting to Databases with TDatabase

TDatabase sets up a persistent connection to a database, especially a remote database requiring a user login and password. **TDatabase** is especially important because it permits control over database transaction processing with the BDE when connected to a remote SQL database server. Use **TDatabase** when a BDE-based database application requires:

- Persistent database connections
- Customized database server logins
- Transaction control
- Application-specific BDE aliases

To connect to databases with TDatabase

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. Associate a database component with a session.
4. Identify the database.
5. Open a connection using TDatabase.

To associate a database component with a session

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. In the **Object Inspector**, set the SessionName property of the **TSession** component.

   SessionName is set to “Default," which means it is associated with the default session component that is referenced by the global Session variable.

4. Add a **TSession** component for each session if you use multiple sessions.
5. Set the SessionName property of the **TDatabase** component to the SessionName property of the **TSession** component to associate your dataset with a session component.
6. Read the Session property to access the session component with which the database is associated at runtime.

   If SessionName is blank or “Default," the Session property references the same **TSession** instance referenced by the global Session variable.

Session enables applications to access the properties, methods, and events of a database component’s parent session component without knowing the session’s actual name. If you are using an implicit database component, the session for that database component is the one specified by the dataset’s SessionName property.

To identify the database

1. In the drop-down lists for dataset components, specify the alias name or the name of an existing BDE alias for a database component.

Note: This clears any value already assigned to DriverName. Alternatively, you can specify a driver name instead of an alias when you create a local BDE alias for a database component using the DatabaseName property. Specifying the driver name clears any value already assigned to AliasName. To provide your own name for a database connection, set the DatabaseName. To specify a BDE alias at designtime, assign a BDE driver.

2 Create a local BDE alias.

3 Double-click a database component.

The **Database** editor opens.

4 In the **Name** edit box in the properties editor, enter the same name as specified by the DatabaseName property.

5 In the **Alias name** combo box, enter an existing BDE alias name or choose from existing aliases in the drop-down list.

6 To create or edit connection parameters at designtime, do one of the following:

- Use the Database Explorer or BDE Administration utility.
- Double-click the Params property in the **Object Inspector** to invoke the **Value List** editor.
- Double-click a database component in a data module or form to invoke the **Database** editor.

Note: All of these methods edit the Params property for the database component. When you first invoke the **Database Properties** editor, the parameters for the BDE alias are not visible. To see the current settings, click **Defaults**. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click **Clear**. Changes you make take effect only when you click **OK**.

To open a connection using TDatabase

1 In the Params property of a **TDatabase** component, configure the ODBC driver for your application.

2 To connect to a database using **TDatabase**, set the Connected property to **True** or call the Open method.

Note: Calling TDatabase. Rollback does not call TDataSet. Cancel for any data sets associated with the database.

Connecting to the Application Server using DataSnap Components

A client application uses one or more connection components in the **DataSnap** category of the **Tool Palette** to establish and maintain a connection to an application server.

To connect to the application server using DataSnap components

1. Identify the protocol for communicating with the application server.
2. Locate the server machine.
3. Identify the application server on the server machine.
4. If you are not using SOAP, identify the server using the ServerName or ServerGUID property.
5. Manage server connections.

Debugging dbExpress Applications using TSQLMonitor

While you are debugging your database application, you can monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the dbExpress driver).

To debug dbExpress applications

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. To monitor SQL commands, from the **dbExpress** category of the **Tool Palette**, drag a **TSQLMonitor** component to the form.
4. Set the SQLConnection property of the **TSQLMonitor** to the **TSQLConnection** component.
5. Set the Active property of the **TSQLMonitor** to **True**.

To use a callback to monitor SQL commands

1. Use the SetTraceCallbackEvent method of the **TSQLConnection** component.
2. Set the parameters, CallType and CBInfo.

   The callback returns a value of type CBRType, typically cbrUSEDEF.

The dbExpress driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

Warning: Do not call SetTraceCallbackEvent if the **TSQLConnection** object has an associated **TSQLMonitor** component. **TSQLMonitor** uses the callback mechanism to work, and **TSQLConnection** can only support one callback at a time.

Executing the Commands using TSQLDataSet

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements. The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it, but pass the command to the server for execution.

To execute commands

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. Specify the command to execute.
5. Execute the command.
6. Create and modify server metadata.

To specify the command to execute

1. Set the CommandType and CommandText properties in the **Object Inspector** to specify the command for a **TSQLDataSet**.
2. Set the SQL property in the **Object Inspector** to specify the SQL statement to pass to the server for a **TSQLQuery** .
3. Set the StoredProcName property in the **Object Inspector** to specify the name of the stored procedure to execute for a **TSQLStoredProc** .

To execute the command

1. If the dataset is an instance of a **TSQLDataSet** or a **TSQLQuery**, call the ExecSQL method.
2. If the dataset is an instance of a **TSQLStoredProc**, call the ExecProc method.

Tip: If you are executing the query or stored procedure multiple times, it is a good idea to set the Prepared property to **True**.

To create and modify server metadata

1. To create tables in a database, use the CREATE TABLE statement.
2. To create new indexes for those tables, use the CREATE INDEX statement.
3. To add various metadata objects, use CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE statements.
4. To delete any of the above metadata objects, use DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

5 To change the structure of a table, use the ALTER TABLE statement.

Fetching the Data using TSQLDataSet

To fetch the data

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. To fetch the data for a unidirectional dataset, do one of the following:

   - In the **Object Inspector**, set the Active property to **True**.
   - Call the Open method at runtime.

   Tip: Use GetMetadata property to selectively fetch metadata on a database object. Set GetMetadata to False if you are fetching a dataset for read-only purposes.
5. Set its Prepared property to **True** to prepare the dataset explicitly.
6. Call the NextRecordSet method to fetch multiple sets of records.

Note: NextRecordSet returns a newly created TCustomSQLDataSet component that provides access to the next set of records. That is, the first time you call NextRecordSet, it returns a dataset for the second set of records. Calling NextRecordSet returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, NextRecordSet does not return anything.

Managing Database Sessions Using TSession

A session provides global connection over a group of database components. A default **TSession** component is automatically created for each database application. You must use **TSession** component only if you are creating a multithreaded database application. Each database thread requires its own session components.

To manage database sessions

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. Activate a session.
4. Specify default database connection behavior.
5. Manage database connections.
6. Work with password-protected Paradox and dBASE tables.
7. Work with BDE aliases.
8. Retrieve information about a session.
9. Create, Name, and Manage additional sessions.

Specifying the Data to Display using TSQLDataSet

To specify the data to display

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. For **TSQLDataSet**, specify the type of unidirectional dataset by CommandType property in the **Object Inspector**.
5. Specify whether information comes from results of query, a database table, or a stored procedure.

To display results from a query

1. Set the CommandType property to ctQuery for a **TSQLDataSet**.
2. For **TSQLQuery**, drag a **TSQLQuery** component from the **Tool Palette** to the form.
3. Set the SQL property to the query you want to assign.
4. Select **TSQLDataSet**.
5. Click the CommandText property.

   The **CommandText Editor** opens.
6. In the **CommandText Editor**, set the SQL property to the text of the query statement.

Note: When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. SQL defines queries such as UPDATE queries that perform actions on the server but do not return records.

To display records in a table

1. In the **Object Inspector**, set the CommandType property to ctTable.

   TSQLDataSet generates a query based on the values of two properties: CommandText that specifies the name of the database table that the **TSQLDataSet** object should represent and SortFieldNames that lists the names of any fields to use to sort the data, in the order of significance
2. Drag a **TSQLTable** component to the form.
3. In the **Object Inspector** , set the TableName property to the table you want.
4. Set the IndexName property to the name of an index defined on the server or set the IndexFieldNames property to a semicolon-delimited list of field names to specify the order of fields in the dataset.

To display the results of a stored procedure

1. In the **Object Inspector**, set the CommandType property to ctStoredProc.
2. Specify the name of the stored procedure as the value of the CommandText property.
3. Set the StoredProcName property to the name of the stored procedure for **TSQLStoredProc**.

Note: After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure.

Specifying the Provider using TLocalConnection or TConnectionBroker

Client datasets are specialized datasets that hold all the data in memory. They use a provider to supply them with data and apply updates when they cache updates from a database server or another dataset, represent the data in an XML document, and store the data in the client portion of a multi-tiered application.

To specify the provider

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. From the **DataSnap** category of the **Tool Palette**, drag a **TConnectionBroker** component to the form if the provider is on a remote application server.
4. In the **Object Inspector**, set the ConnectionBroker property of your client dataset to the **TConnectionBroker** component to the form.
5. From the **DataSnap** category of the **Tool Palette**, drag a **TLocalConnection** component to the form if the provider is in the same application as the client dataset.
6. Set the RemoteServer property of your client dataset to the **TLocalConnection** component to the form.

Using BDE

To use BDE

1. Choose **File** ▸ **New** ▸ **Other**.
   The **New Items** dialog box opens.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.
   The **Windows Designer** displays.
3. From the **BDE** category of the **Tool Palette**, drag a **TTable** component to the form.
   This will encapsulate the full structure of data in an underlying database table.
4. From the **BDE** category of the **Tool Palette**, drag a **TQuery** component to the form.
   This will encapsulate an SQL statement and enables applications to access the resulting records.
5. From the **BDE** category of the **Tool Palette**, drag a **TStoredProc** component to the form.
   This will execute a stored procedure that is defined on a database server.
6. From the **BDE** category of the **Tool Palette**, drag a **TBatchMove** component to the form.
   This will copy a table structure or its data.
7. From the **BDE** category of the **Tool Palette**, drag a **TUpdateSQL** component to the form.
   This will provide a way to update the underlying datasets.

Using DataSnap

A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

Multi-tiered applications use the components on the **DataSnap** category in the **Tool Palette**. DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

To build multi-tiered database applications using DataSnap

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. From the **DataSnap** category of the **Tool Palette**, drag a **TDCOMConnection** component to the form.

   This will establish a DCOM connection to a remote server in a multi-tiered database application.

4. From the **DataSnap** category of the **Tool Palette**, drag a **TSocketConnection** component to the form.

   This will establish a TCP/IP connection to a remote server in a multi-tiered database application.

5. From the **DataSnap** category of the **Tool Palette**, drag a **TSimpleObjectBroker** component to the form.

   This will locate a server for a connection component from a list of available application servers.

6. From the **DataSnap** category of the **Tool Palette**, drag a **TWebConnection** component to the form.

   This will establish an HTTP connection to a remote server in a multi-tiered database application.

7. From the **DataSnap** category of the **Tool Palette**, drag a **TConnectionBroker** component to the form.

   This will centralize all connections to the application server so that applications do not need major rewriting when changing the connection protocol.

8. From the **DataSnap** category of the **Tool Palette**, drag a **TSharedConnection** component to the form.

   This will connect to a child remote data module when the application server is built using multiple remote data modules.

9. From the **DataSnap** category of the **Tool Palette**, drag a **TLocalConnection** component to the form.

   This will provide access to IAppServer methods that would otherwise be unavailable, and make it easier to scale up to a multi-tiered application at a later time. It acts like a connection component for providers that reside in the same application.

Using dbExpress

To build a database applications using dbExpress

1. Connect to the database server and configure a TSQL connection.
2. Specify the data to display.
3. Fetch the data.
4. Execute the commands.
5. Access the schema information.
6. Debug dbExpress application using **TSQLMonitor**.
7. Use **TSQLTable** to represent a table on a database server that is accessed via **TSQLConnection**.
8. Use **TSQLQuery** to execute an SQL command on a database server that is accessed via **TSQLConnection**.
9. Use **TSQLStoredProc** to execute a stored procedure on a database server that is accessed via **TSQLConnection**.

Using TBatchMove

TBatchMove copies a table structure or its data. It can be used to move entire tables from one database format to another.

To use TBatchMove

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. Create a batch move component.
4. Specify a batch move mode.
5. Map data types.
6. Execute a batch move.
7. Handle batch move errors.

Using TQuery

TQuery is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records.

To use TQuery

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. Associate the dataset with database and session connections.
4. Create heterogeneous queries.
5. Obtain an editable result set.
6. Update read-only result sets.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. Set the DatabaseName property of the **TDatabase** component to associate a BDE-enabled dataset with a database.

   For the **TDatabase** component, database name is the value of the DatabaseName property of the database component.

4. Specify a BDE alias as the value of DatabaseName if you want to use an implicit database component and the database has a BDE alias.

   Note: A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on).

5. In the **Object Inspector**, set the DatabaseName to specify the directory where the database tables are located if you want to use an implicit database component for a Paradox or dBASE database.
6. Use the default session to control all database connections in your application.
7. Set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component .

Note: Whether you use the default session or explicitly specify a session using the SessionName property, you can access the session associated with a dataset by reading the DBSession property. If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.

To create mixed queries

1. Define separate BDE aliases for each database accessed in the query using the BDE Administration tool or the SQL explorer.
2. Leave the DatabaseName property of the **TQuery** component blank.

   The names of the databases used will be specified in the SQL statement.

3 Set the SQL property to the SQL statement you want to execute.

4 Precede each table name in the statement with the BDE alias for the database of the table, enclosed in colons. This whole reference is then enclosed in quotation marks.

5 Set the Params property to any parameters for the query.

6 Write a Prepare method to prepare the query for execution prior to executing it for the first time.

7 Write an Open or ExecSQL method depending on the type of query you are executing.

8 Use a **TDatabase** component as an alternative to using a BDE alias to specify the database in a mixed query.

9 Configure the **TDatabase** to the database, set the TDatabase. DatabaseName to an unique value, and use that value in the SQL statement instead of a BDE alias name.

To obtain an editable result set

1 Set RequestLive property of the **TQuery** component to **True**.

2 If the query contains linked fields, treat the result set as a read-only result set, and update it.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either a read-only result set for queries made against Paradox or dBASE, or an error code for SQL queries made against a remote server.

To update read-only result sets

1 If all updates are applied to a single database table, indicate the underlying table to update in an OnGetTableName event handler.

2 Set the query's UpdateObject property to the **TUpdateSQL** object you are using to have more control over applying updates.

3 Set the DeleteSQL, InsertSQL, and ModifySQL properties of the update object to the SQL statements that perform the appropriate updates for your query's data.

If you are using the BDE to cache updates, you must use an update object.

Using TSimpleDataSet

TSimpleDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

To use TSQLStoredProc

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSimpleDataSet** component to the form.
2. Set its Name property to a unique value appropriate to your application.
3. From the **dbExpress** section of the **Tool Palette**, drag a **TSQLConnection** component on the form.
4. Select **TSimpleDataSet** component. Set the Connection property to **TSQLConnection** component.
5. To fetch data from the server, do any of the following:
   - Set CommandType to ctQuery and set CommandText to an SQL statement you want to execute on the server.
   - Set CommandType to ctStoredProc and set CommandText to the name of the stored procedure you want to execute.
   - Set CommandType to ctTable and set CommandText to the name of the database tables whose records you want to use.
6. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the form.
7. Set the DataSet property of the data source component to the **TSimpleDataSet** object.
8. To activate the dataset, use the Active property or call the Open method.
9. If you executed a stored procedure, use the Params property to retrieve any output parameters.

Using TSimpleObjectBroker

If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system.

To use TSimpleObjectBroker

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.

3. From the **DataSnap** category of the **Tool Palette**, choose the connection component depending on the kind of connection you want.
4. From the **Tool Palette**, drag a **TSimpleObjectBroker** to the form.
5. In the **Object Inspector**, set the ObjectBroker property of the connection component that you chose in Step 3 to use this broker.

Warning: Do not use the ObjectBroker property with SOAP connections.

Using TSQLQuery

TSQLQuery represents a query that is executed using dbExpress. **TSQLQuery** can represent the results of a SELECT statement or perform actions on the database server using statements such as INSERT, DELETE, UPDATE, ALTER TABLE, and so on. You can add a **TSQLQuery** component to a form at design time, or create one dynamically at runtime.

To use TSQLQuery

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLQuery** component to the form.
2. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
3. Set the SQLConnection property.
4. Click the ellipsis button next to the SQL property of the **TSQLQuery** component.

   The **String List** editor opens.
5. In the **String List** editor, type the query statement you want to execute.
6. If the query data is to be used with visual data controls, add a data source component to the form.
7. Set the DataSet property of the data source component to the query-type dataset.
8. To activate the query component, set the Active property to **True** or call the Open method at runtime.

Using TSQLStoredProc

TSQLStoredProc represents a stored procedure that is executed using dbExpress. **TSQLStoredProc** can represent the result set if the stored procedure returns a cursor. You can add a **TSQLStoredProc** component to a form at design time, or create one dynamically at runtime.

To use TSQLStoredProc

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLStoredProc** component to the form.
2. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
3. Set the SQLConnection property.
4. Set the StoredProcName property to specify the stored procedure to execute.
5. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the form.
6. Set the DataSet property of the data source component to the stored procedure-type dataset.
7. Provide input parameter values for the stored procedure, if necessary.
8. To execute the stored procedure that returns a cursor, use the Active property or call the Open method.
9. Process any results.

Using TSQLTable

TSQLTable represents a database table that is accessed using dbExpress. **TSQLTable** generates a query to fetch all of the rows and columns in a table you specify. You can add a **TSQLTable** component to a form at designtime, or create one dynamically at runtime.

To use TSQLTable

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog displays.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLTable** component to the form.
4. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
5. Set the SQLConnection property
6. Set the TableName property to the name of the table in the database.
7. Add a data source component to the form.
8. Set the DataSet property of the data source component to the the name of the dataset.

Using TStoredProc

TStoredProc is a stored procedure-type dataset that executes a stored procedure that is defined on a database server.

To use TStoredProc

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. Associate a dataset with database and session connections.
4. Bind the parameters.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. To associate a BDE-enabled dataset with a database, set the DatabaseName property.

   For **TDatabase** component, database name is the value of the DatabaseName property of the database component.
3. Drag a **TSession** component to the form.
4. To control all database connections in your application, use the default session.
5. In the **Object Inspector**, set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component.

Note: If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.

To bind parameters

1. From the **BDE** category of the **Tool Palette**, drag a **TStoredProc** component to the form.
2. Set the ParamBindMode property to default pbByName to specify how parameters should be bound to the parameters on the server.
3. View the stored procedure source code of a server in the SQL Explorer if you want to set ParamBindMode to pbByNumber.
4. Determine the correct order and type of parameters.
5. Specify the correct parameter types in the correct order.

Note: Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set ParamBindMode to pbByNumber.

Using TTable

TTable is a table-type dataset that represents all of the rows and columns of a single database table.

To use TTable

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**.

   The **Windows Designer** displays.
3. Associate the dataset with the database and session connections.
4. Specify the table type for local tables and control read/write access to local tables.
5. Specify a dBASE index file.
6. Rename local tables.
7. Import data from another table.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. To associate a BDE-enabled dataset with a database, in the **Object Inspector**, set the DatabaseName property of the **TDatabase** component .

   For a **TDatabase** component, the database name is the value of the DatabaseName property of the database component.
4. Use the default session to control all database connections in your application.
5. Set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component.

If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.

To specify the TableType and control read/write access

1. From the **BDE** category of the **Tool Palette**, drag a **TTable** component to the form.
2. In the **Object Inspector**, set the TableType property if an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables.

   BDE uses the TableType property to determine the table's type.
3. Set TableType to ttDefault if your local Paradox, dBASE, and ASCII text tables use the file extensions like, .DB, . DBF, and .TXT.
4. For other extensions, set TableType to ttParadox for Paradox, ttDBase for dBASE, ttFoxPro for FoxPro, and ttASCII for Comma-delimited ASCII text respectively.
5. Set the table component's Exclusive property to **True** before opening the table to gain sole read/write access.

   Note: If the table is already in use when you attempt to open it, exclusive access is not granted. You can attempt to set Exclusive on SQL tables, but some servers do not support exclusive table-

level locking. Others may grant an exclusive lock, but permit other applications to read data from the table.

To specify a dBASE index file

1. Set the IndexFiles property to the name of the non-production index file or list the files with a .NDX extension.
2. Specify one index in the IndexName property to have it actively sorting the dataset.
3. At designtime, click the ellipsis button in the IndexFiles property.

   The **Index Files** editor opens.
4. To add a non-production index file or file with .NDX extension, click the **Add** button in the **Index Files** dialog and select the file from the **Open** dialog.

   Note: For each non-production index file or .NDX file, repeat Steps 3 and 4.
5. After adding all desired indexes, click the **OK** button in the **Index Files** editor.

Note: To do steps 3-5 at runtime, access the IndexFiles property using properties and methods of string lists.

To rename local tables

1. To rename a Paradox or dBASE table at design time, right-click the table component.

   A drop-down context menu opens.
2. From the context menu, select **Rename Table**.
3. To rename a Paradox or dBASE table at runtime, call the table's RenameTable method.

To import data from another table

1. Use the BatchMove method of a table component to import data, copy, update, append records from another table into this table, or delete records from a table.
2. Set the name of the table from which to import data, and a mode specification that determines which import operation to perform.

Using TUpdateSQL to Update a Dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not “live”, it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates.

To update a dataset using an update object

1. From the **Tool Palette**, add a **TUpdateSQL** component to the same form as the BDE-enabled dataset.
2. In the **Object Inspector**, set the UpdateObject property of the BDE-enabled dataset component’s to the **TUpdateSQL** component in the form.
3. Set the ModifySQL, InsertSQL, and DeleteSQL properties of the update object to specify the SQL statements needed to perform updates.
4. Close the dataset.
5. Set the dataset component’s CachedUpdates property to **True** or link the dataset to the client dataset using a dataset provider.
6. Reopen the dataset.
7. Create SQL statements for update components.
8. Use multiple update objects.
9. Execute the SQL statements.

Interoperable Applications Procedures

Using COM Wizards

Developer Studio 2006 provides wizards that help you create COM projects and COM objects. These wizards are available for both Delphi, and C++ projects. The following COM wizards are provided:

- ActiveX Library
- COM Object
- Type Library
- Active Form
- Active Server Object
- Automation Object
- COM+ Event Object
- COM+ Subscription Object
- Property Page

To use a COM wizard

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog box displays.

2. In the **Item Categories** tree, click the **ActiveX** folder.

   The wizards available are shown in the right-hand pane of the **New Items** dialog.

   COM wizards are available for both C++ and Delphi projects.

3. Double-click the wizard you wish to use.

Note: If your application implements more than one COM object, you should specify the same instancing for all of them.

Reporting Procedures

Adding Rave Reports to Developer Studio 2006

Rave Reports offers a powerful set of tools for building reports and including them in your applications. Rave Reports are installed in a \RaveReports subdirectory in your installation directory. To make the Rave Reports more easily accessible, add the command executable to your **Tools** menu.

To add a Rave Reports command to the Tools menu

1. Choose **Tools** ▸ **Configure Tools**.

   This displays the **Tool Options** dialog box.
2. Click **Add**.

   This displays the **Tool Properties** dialog box.
3. Type Rave Reports in the **Title** text box.
4. Click the **Browse** button.
5. Browse to the \RaveReports subdirectory in your Developer Studio 2006 installation directory.
6. Select the **Rave.exe** icon.
7. Click **OK**.

   This adds the path for the program and the working directory to the **Tool Properties** dialog box.
8. Click **OK**
9. Click **Close**.

   This adds the command to your **Tools** menu that will initiate a Rave Reports session. Refer to the Rave Reports online Help for information on how to build and integrate report objects.

VCL Procedures

Adding and Sorting Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with Button, Label, and TListBox controls.
2. Write the code to add and sort strings.
3. Run the application.

To create a VCL Form with Button, Label, and ListBox controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays.
2. From the **Standard** category of the **Tool Palette**, place a TButton, TLabel, and TListBox component on the form.

To write the copy stream procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the TForm1.Button1Click event handler block.
3. For Delphi, place the cursor before the `begin` reserved word and press ENTER.

   This creates a new line above the code block.
4. Type the following variable declarations:

```
[Delphi]
var
  MyList: TStringList;
  Index: Integer;
```

```
[C++]
TStringList *MyList;
int Index;
```

5. Insert the cursor within the code block, and type the following code:

```
[Delphi]
MyList := TStringList.Create;
try
  MyList.Add('Animals');
  MyList.Add('Flowers');

  MyList.Add('Cars');

  MyList.Sort;
    if MyList.Find('Flowers', Index) then
    begin
      ListBox1.Items.AddStrings(MyList);
      Label1.Caption := 'Flowers has an index value of ' + IntToStr(Index);
    end;
```

```
finally
    MyList.Free;
end;
```

```
[C++]
MyList = new TStringList();
try {
  MyList->Add( "Animals" );
  MyList->Add( "Flowers" );
  MyList->Add( "Cars" );
  MyList->Sort();
  if( MyList->Find( "Flowers", Index ) {
    ListBox1->Items->AddStrings( MyList );
    Label1->Caption = "Flowers has an index of " +
                      IntToStr( Index );
  }
} __finally {
  MyList->Free();
}
```

Note: Find will only work on sorted lists. Use IndexOf on unsorted lists.

To run the application

1 Save your project files; then choose **Run** ▶ **Run** to build and run the application.

The form displays with the controls.

2 Click the Button.

The strings 'Animals', 'Cars', and 'Flowers' display alphabetically in a list in the ListBox. The Label caption displays the message string: 'Flowers has an index value of 2.'

Avoiding Simultaneous Thread Access to the Same Memory

Use these basic techniques to prevent other threads from accessing the same memory as your thread:

Lock objects.

Use critical sections.

Use a multi-read exclusive-write synchronizer

To lock objects

1. For objects such as canvas that have a `Lock` method, call the `Lock` method, as necessary, to prevent other objects from accessing the object, and call `Unlock` when locking is no longer required.
2. Call `TThreadList.LockList` (Delphi) or `TThreadList::LockList()` (C++) to block threads from using the list object `TThreadList`, and call `TThreadList.UnlockList` when locking is no longer required.

Note: You can safely make calls to `TCanvas.Lock` and `TThreadList.LockList`.

To use a critical section

1. Create a global instance of `TCriticalSection`.
2. Call the `Acquire` method to lock out other threads while accessing global memory.
3. Call the `Release` method so other threads can access the memory by calling `Acquire`.

   The following code has a global critical section variable `LockXY` that blocks access to the global variables X and Y. To use X or Y, a thread must surround that use with calls to the critical section such as shown here:

```
[Delphi]
LockXY.Acquire;
try
      X := X + 1;
  Y := sin(X);
finally
  LockXY.Release
end;
```

```
[C++]
LockXY->Acquire();
try {
  x++;
  y = sin( x );
} __finally {
  LockXY->Release();
}
```

Warning: Critical sections only work if every thread uses them to access global memory. Otherwise, problems of simultaneous access can occur.

To use the multi-read exclusive-write synchronizer

1. Create a global instance of `TMultiReadExclusiveWriteSynchronizer` that is associated with the global memory you want to protect.
2. Before any thread reads from the memory, it must call `BeginRead`.
3. At the completion of reading memory, the thread must call `EndRead`.
4. Before any thread writes to the memory, it must call `BeginWrite`.
5. At the completion of writing to the memory, the thread must call `EndWrite`.

Warning: The multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Otherwise, problems of simultaneous access can occur.

Building a Multithreaded Application

These are the essential steps to building a VCL Forms multithreaded application with a thread object using Developer Studio 2006.

To drop a component on a form

1. Create a VCL form with a defined thread object.
2. Optionally initialize the thread.
3. Write the thread function.
4. Optionally write the cleanup code.

Building a VCL Forms "Hello world" Application

Though simple, the Windows Forms "Hello world" application demonstrates the essential steps for creating a VCL Forms application. The application uses a VCL Form, a control, an event, and will display a dialog in response to a user action.

Creating the "Hello world" application consists of the following steps:

1. Create a VCL Form with a button control.
2. Write the code to display "Hello world" when the button is clicked.
3. Run the application.

To create a VCL Form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. Click the VCL form to display the form view.
3. From the **Standard** page of the Tool Palette, place a TButton component on the form.

To display the "Hello world" string

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `Button1Click` event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press ENTER. This creates a new line above the code block.

   For C++, place the cursor after the opening brace (`{`) and press ENTER.
4. For Delphi, insert the cursor on the new line created, and type the following variable declaration:

   ```
   [Delphi]
   var s: string;
   ```

   For C++, enter the following code:

   ```
   [C++]
   AnsiString s;
   ```
5. For Delphi, insert the cursor within the code block and type the following code:

   ```
   s:= 'Hello world!';
   ShowMessage(s);
   ```

   For C++, enter the following code:

```
[C++]
s = “Hello world!”;
ShowMessage( s );
```

To run the "Hello world" application

1. Choose **Run** ▶ **Run** to build and run the application.
   The form displays with a button called **Button1**.
2. Click **Button1**.
   A dialog box displays the message "Hello World!"
3. Close the VCL form to return to the IDE.

Building a VCL Forms ADO Database Application

The following procedure describes how to build an ADO database application.

Building a VCL ADO application consists of the following major steps:

1. Set up the database connection.
2. Set up the dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add an ADO connection component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **ADO** page of the **Tool Palette**, place an ADOConnection component on the form.
3. Double-click the ADOConnection component to display the **ConnectionString** dialog.
4. If necessary, select **Use Connection String**; then click the **Build** button to display the **Link Properties** dialog.
5. On the **Provider** page of the dialog, select **Microsoft Jet 4.0 OLE DB Provider**; then click the **Next** button to display the **Connections** page.
6. On the **Connections** page, click the ellipsis button to browse for the dbdemos.mdb database. The default path to this database is C:\Program Files\Common Files\Borland Shared\Data.
7. Click **Test Connection** to confirm the connection. A dialog appears, indicating the status of the connection.
8. Click **OK** to close the **Data Link Properties** dialog. Click **OK** to close the **ConnectionString** dialog.

To set up the dataset

1. From the **ADO** page, place an ADODataSet component at the top of the form.
2. In the **Object Inspector**, select the Connection property drop-down list. Set it to ADOConnection1.
3. Set the CommandText property to an SQL command, for example, Select * from orders.

   You can either type the Select statement in the **Object Inspector** or click the ellipsis to the right of CommandText to display the **CommandText Editor** where you can build your own query statement.

   Tip: If you need additional help while using the **CommandText Editor**, click the **Help** button.
4. Set the Active property to True to open the dataset.

   You are prompted to log in. Use admin for the username and no password.

To add the provider

1. From the **Data Access** page, place a DataSetProvider component at the top of the form.
2. In the **Object Inspector**, select the DataSet property drop-down list. Set it to ADODataSet1.

To add client dataset

1. From the **Data Access** page, place a ClientDataSet component to the right of the DataSetProvider component on the form.
2. In the **Object Inspector**, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application.

   A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1. In the **Tool Palette** on the **Data Access** page, place a DataSource component to the right of the ClientDataSet on the form.
2. In the **Object Inspector**, select the DataSet property drop-down. Set it to ClientDataSet1.

To connect a DataGrid to the DataSet

1. In the **Tool Palette** on the **Data Controls** page, place a DBGrid component on the form.
2. In the **Object Inspector**, select the DataSource property drop-down. Set the data source to DataSource1.
3. Choose **Run** ▸ **Run**.
4. You are prompted to log in. Enter admin for the username and no password.

   The application compiles and displays a VCL form with a DBGrid.

Building a VCL Forms Application

The following procedure illustrates the essential steps to building a VCL Forms application using Developer Studio 2006.

To create a VCL Form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Tool Palette**, place components onto the form to create the user interface.
3. Write the code for the controls.

To associate code with a control

1. Double-click the component on the form to which you want to apply logic. The **Code Editor** displays, cursor in place within the event handler block.
2. Code your application logic.
3. Save and compile the application.

Building a VCL Forms Application with Decision Support Components

Creating a form with tables and graphs of multidimensional data consists of the following major steps:

1. Create a VCL form.
2. Add a decision query and dataset.
3. Add a decision cube.
4. Add a decision source.
5. Optionally add a decision pivot.
6. Add one or more decision grids and graphs.
7. Set the active property of the decision query (or alternate dataset component) to True.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays.
2. If necessary, click Form1 to make it the active window.

To add a decision dataset

1. From the **Decision Cube** page on the **Tool Palette**, add a DecisionQuery component to the top of the form.

   Tip: Place non-visual components such as this one in the top left corner of the form to keep them out of the way of visual components you will be adding.
2. Right-click the DecisionQuery component, and select **Decision Query Editor...**.

   The **Decision Query Editor** displays.
3. On the **Dimensions/Summary** tab, select the BCDEMOS database from the **Database:** drop-down list.
4. From the **Table:** drop-down, select the parts.db table.

   The **List of Available Fields:** listbox displays the fields in the parts.db table.
5. Use CTRL+Click to select the PartNo, OnOrder, and Cost fields; then click the right-arrow button next to the **Dimensions:** listbox.

   PartNo, OnOrder, and Cost display in the listbox.
6. Select the OnOrder field; then click the right-arrow button next to the **Summaries:** listbox and select count from the pop-up that displays.

   COUNT(OnOrder) displays in the **Summaries:** listbox.
7. Select the Cost field in the **List of Available Fields:** listbox; then click the right-arrow button next to the **Summaries:** listbox and select sum from the pop-up that displays.

   SUM(Cost) displays in the **Summaries:** listbox.
8. Click **OK** to close the **Decision Query Editor**.

Note: When you use the Decision Query Editor, the query is initially handled in ANSI-92 SQL syntax and then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays

only ANSI standard SQL. The dialect translation is automatically assigned to the TDecisionQuery's SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather than the SQL property.

To add a decision cube

1. From the **Decision Cube** page on the **Tool Palette**, add a decision cube component to the top left corner of the form.
2. In the **Object Inspector**, select DecisionQuery1 from the drop-down list next to the decision cube's DataSet property.

To add a decision source

1. From the **Decision Cube** page on the **Tool Palette**, add a decision source component to the top left corner of the form.
2. In the **Object Inspector**, select DecisionCube1 from the drop-down list next to the decision source's DecisionCube property.

To add a decision pivot

1. From the **Decision Cube** page on the **Tool Palette**, add an optional DecisionPivot component to the top of the form.

   Tip: The decision pivot displays in the final application window. Place it to the right of the nonvisual components.

2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision pivot's DecisionSource property.

To create a decision grid

1. From the **Decision Cube** page on the **Tool Palette**, add a decision grid component to the form just beneath the decision pivot.
2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision grid's DecisionSource property.

To create a decision graph

1. From the **Decision Cube** page on the **Tool Palette**, add a decision graph component to the form just beneath the decision grid.
2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision graph's DecisionSource property.

To run the application

1. In the **Object Inspector**, select True from the Active property drop-down.

   The visual decision graph, grid, and pivot components display data.

2. Choose **Run** ▸ **Run** to run the application.

The application runs and displays the decision support components.

3 Use the decision pivot to update, as desired, the data displayed in the grid and graph.

Building a VCL Forms dbExpress Database Application

The following procedure describes how to build a dbExpress database application.

Building a VCL Forms dbExpress application consists of the following major steps:

1. Set up the database connection.
2. Set up the unidirectional dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add a dbExpress connection component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **dbExpress** page of the **Tool Palette**, place a TSQLConnection component on the form.
3. Double-click the TSQLConnection component to display the **Connection Editor**.
4. In the **Connection Editor**, set the **Connection Name** field to IBConnection.
5. In the **Connections Setting** box, specify the path to the InterBase database file called employee.gdb in the Database field.

   By default, the file is located in C:\Program Files\Common Files\Borland Shared\Data.
6. Accept the value in the **User_Name** field (sysdba) and **Password** field (masterkey).
7. Click **OK** to close the **Connection Editor** and save your changes.

To set up the unidirectional dataset

1. In the **Tool Palette** on the **dbExpress** page, place a TSQLDataSet component at the top of the form.
2. In the **Object Inspector**, select the SQLConnection property drop-down list. Set it to TSQLConnection1.
3. Set the CommandText property to an SQL command, for example, Select * from SALES.

   You are prompted to log in. Use the masterkey password.

   For the SQL command, you can either type a Select statement in the Object Inspector or click the ellipsis to the right of CommandText to display the **CommandText Editor** where you can build your own query statement.

   Tip: If you need additional help while using the **CommandText Editor**, click the **Help** button.
4. In the **Object Inspector**, set the Active property to True to open the dataset.

To add the provider

1. In the **Tool Palette** on the **Data Access** page, place a TDataSetProvider component at the top of the form.
2. In the **Object Inspector**, select the DataSet property drop-down list. Set it to SQLDataSet1.

To add client dataset

1. In the **Tool Palette** on the **Data Access** page, place a TClientDataSet component to the right of the TDataSetProvider component on the form.
2. In the **Object Inspector**, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application.

   A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1. In the **Tool Palette** on the **Data Access** page, place a TDataSource component to the right of the TClientDataSet on the form.
2. In the **Object Inspector**, select the DataSet property drop-down. Set it to ClientDataSet1.

To connect a DataGrid to the DataSet

1. In the **Tool Palette** on the **Data Controls** page, place a TDBGrid component on the form.
2. In the **Object Inspector**, select the DataSource property drop-down. Set the data source to DataSource1.
3. Choose**Run** ▸ **Run**.

   You are prompted to enter a password. Enter masterkey. If you enter an incorrect password or no password, the debugger throws an exception.

   The application compiles and displays a VCL form with a DBGrid.

Building a VCL Forms MDI Application Using a Wizard

The VCL Forms MDI application wizard automatically creates a project that includes the basic files for an MDI application. In addition to the Main source file, the wizard creates unit files for child and about box windows, along with the supporting forms files and resources.

To create a new MDI application using a wizard

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-click the **MDI Application** icon.
   The **Browse to Folder** dialog box is displayed.
2. Navigate to the folder in which you want to store the files for the project.
3. Click **OK**.
4. Choose **Run** ▸ **Run** to compile and run the application.
5. Try commands that are automatically set up by the MDI Application wizard.

Building a VCL Forms MDI Application Without Using a Wizard

The basic steps to create a new MDI application with a child window without using a wizard are

1. Create a main window form (MDI parent window).
2. Create a child window form.
3. Have the main window create the child window under user control.
4. Write the event handler code to close the child window.
5. Create the main menu and commands.
6. Add the event handlers for the commands.
7. Compile and run the application.

To create the main window form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. In the **Object Inspector**, set the FormStyle property to fsMDIForm.
3. Enter a more descriptive name such as frMain for the Name property.
4. Save the unit file with a more descriptive name, such as uMain.pas (Delphi) or uMain.cpp (C++).

To create a child window

1. Choose **File** ▸ **New** ▸ **Form**
2. In the **Object Inspector**, set the FormStyle property to fsMDIChild.
3. Enter a more descriptive name such as frChild for the Name property.
4. Save the unit file as uChild.pas (Delphi) or uChild.cpp (C++).

To have the main window create the child window

1. Choose **Project** ▸ **Options** ▸ **Forms**.

   The **Project Options** dialog box appears.
2. Select **frChild** from **Auto-create forms** list and click **[>]** to move it to the **Available forms** list and click **OK**.
3. Select the frMain form to activate it; then switch to the Code view.
4. For Delphi, scroll to the `uses` section and add `uChild` to the list.

   For C++, add `#include "uChild.h"` to uMain.h.
5. For Delphi, scroll to the `private` declarations section and enter this procedure declaration:

```
[Delphi]
procedure CreateChildForm(const childName: string);
```

   For C++, add the following function declaration to the private: declarations of `TfrMain`.

```
[C++]
void __fastcall CreateChildForm( const AnsiString childName );
```

6 For Delphi, scroll to the `implementation` section, and enter the code below:

```
[Delphi]
procedure TfrMain.CreateChildForm (const childName: string);
  var Child: TfrChild;
  begin
    Child := TfrChild.Create(Application);
    Child.Caption := childName;
  end;
```

For C++, add the following function definition to uMain.cpp:

```
[C++]
void __fastcall TfrMain::CreateChildForm( const AnsiString childName )
{
        TfrChild *Child = new TfrChild( Application );
        Child->Caption = childName;
}
```

To write the event handler code to close the child window

1 If necessary, activate the frMain form; then select the **Events** tab in the **Object Inspector**.

2 Double-click the OnClose event.

The **Code Editor** displays with the cursor in the `TfrMain.FormClose` (Delphi) or `TfrMain::FormClose` (C++) event handler block.

3 For Delphi, enter the following code:

```
Action := caFree;
```

For C++, enter the following code:

```
[C++]
Action = caFree;
```

To create the main menu and commands

1 From the **Standard** page on the Tool Palette, place a TMainMenu component on the main form.

2 Double-click the TMainMenu component.

The Menu designer (frMain.MainMenu1) displays with the first blank menu item highlighted.

3 In the **Object Inspector** on the **Properties** tab, enter mnFile for the Name property and &File for the Caption property; then press ENTER.

In the Menu designer, File displays as the name of the first menu item.

4 In the Menu designer, select File.

A blank command field displays in the File group. Select the blank command.

5 In the **Object Inspector**, enter mnNewChild for the Name property and &New child for the Caption property; then press ENTER.

   In the Menu designer, New child displays as the name of the first file command, and a blank command field displays just beneath New child.

6 Select the blank command.

7 In the **Object Inspector**, enter mnCloseAll for the Name property and &Close All for the Caption property; then press ENTER.

   In the Menu designer, Close All displays as the name of the second file command.

To add event handlers for the New child and Close All commands

1 If necessary, open the Menu designer and select New child.

2 In the **Object Inspector**, double-click the OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TfrMain.mnNewChildClick` (Delphi) or `TfrMain::mnNewChildClick` (C++) event handler block.

3 For Delphi, enter the following code:

```
CreateChildForm('Child '+IntToStr(MDIChildCount+1));
```

   For C++, enter the following code:

```
[C++]
CreateChildForm( "Child " + IntToStr( MDIChildCount + 1 ) );
```

4 In the Menu designer, select Close All.

5 In the **Object Inspector**, double-click the OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TfrMain.mnCloseAllClick` (Delphi) or `TfrMain::mnCloseAllClick` (C++) event handler block.

6 For Delphi, enter the following code:

```
for i:=0 to MDIChildCount - 1 do
  MDIChildren[i].Close;
```

   For C++, enter the following code:

```
[C++]
for( int i = 0; i < MDIChildCount; i++ ) {
        MDIChildren[i]->Close();
}
```

7 For Delphi, declare the local variable `i`. The first two lines of the event handler code should appear as shown here when you are done:

```
procedure TfrMain.mnCloseAllClick(Sender: TObject);
  var i: integer;
```

Note: The event handler minimizes the child window in the main window. To close the child window, you must add an OnClose procedure to the child form (next).

To close the child window

1. Activate the child form.
2. In the **Object Inspector**, double-click the OnClose event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TfrChild.FormClose` (Delphi) or `TfrChild::FormClose` (C++) event handler block.
3. For Delphi, enter the following statement:

   ```
   Action := caFree;
   ```

   For C++, enter the following statement:

   ```
   [C++]
   Action = caFree;
   ```

To compile and run the MDI application

1. Choose **Run** ▸ **Run**.
2. The application executes, displaying the File command.
3. Choose **File** ▸ **New child** one or more times.

   A child window displays with each New child command.
4. Choose **File** ▸ **Close All**.

   The child windows close.

Building a VCL Forms SDI Application

To create a new SDI application

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-click the **SDI Application** icon.
2. Pick a folder to save the files in and click **OK**.
3. Choose **Run** ▸ **Run** to compile and run the application.

Building a VCL Forms Web Browser Application

Creating the Web browser application consists of the following steps:

1. Create a VCL Form with a button control.
2. Add a TWebBrowser component to the form.
3. Add controls to enter a URL and launch the browser.
4. Write the code to launch the browser when a button is clicked.
5. Run the application.

To create a VCL Form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Internet** page of the **Tool Palette**, place a TWebBrowser component on the form.
3. With the TWebBrowser component selected on the form, drag the handles to adjust the size of the browser window. Leave some space on the form above the TWebBrowser to add a URL entry window.

   If the window is not large enough to display a browser page in its entirety, the TWebBrowser component adds scrollbars when you run the application and launch the browser window.
4. From the **Standard** page of the **Tool Palette**, place a TMemo component on the form.

   With the TMemo component selected on the form, drag the handles to adjust the size to accommodate a user-entered URL.
5. From the **Standard** page of the **Tool Palette**, place a Label component on the form.
6. Select the Label, and in the **Object Inspector**, enter URL: as the Label caption.
7. From the **Standard** page of the **Tool Palette**, place a TButton component on the form.
8. Select the Button, and in the **Object Inspector**, enter OK as the TButton caption.

To code a button click event that launches the browser

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the Button1Click event handler block.
3. Type the following code:

```
[Delphi]
WebBrowser1.Navigate(WideString(Memo1.Text));
```

```
[C++]
WebBrowser1->Navigate( WideString( Memo1->Text ) );
```

To run the application

1. Choose **Run** ▸ **Run** to build and run the application.

2 Enter a URL to a Web page in the memo window; then click the button.
The browser launches in the TWebBrowser window.

Building a Windows "Hello World" Application

The Windows "Hello World" application demonstrates the essential steps for creating a Windows application. The application uses Windows, a control, an event, and will display a dialog in response to a user action.

To create the "Hello world" application

1. Create a Windows form.
2. Create the logic.
3. Run the application.

To create a Windows form

1. Choose **File** ▸ **New** ▸ **Other...**.

   The **New Items** dialog box appears.
2. In the **New Items** dialog box, select **Delphi Projects** or **C++Builder Projects** and double-click **Application**.
3. If necessary, select **Design** view to display the **Form Designer**
4. From the **Tool Palette**, drag a TButton control onto the designer.
5. Select **Properties** tab in **Object Inspector**.
6. With the button control selected, set the button's **Caption** property to Hello World .

To associate code with the button control

1. In the designer, double-click the button control.

   The **Code Editor** appears with the cursor placed in the event handler code block.
2. Code the application logic:

```
[Delphi]
ShowMessage('Hello, Developer!');
```

```
[C++]
ShowMessage("Hello, Developer!");
```

3. Save and compile the application.

To run the "Hello World" application

1. Choose **Run** ▸ **Run**.

   The application compiles and displays a form with the "Hello World" button.
2. Click the "Hello World" button.

   The "Hello, Developer!" message appears in a dialog box.
3. Close the form to return to the IDE.

Building a Windows Application

The following procedure illustrates the essential steps for building a Windows application.

To create a Windows project

1. In the **New Items** dialog, select **Delphi Projects** and double-click **Application**.
   The **Windows Designer** displays.
2. If necessary, select **Design** view.
3. From the **Tool Palette**, drag components onto the designer to create the user interface.
4. Associate logic with controls.

To associate code with a control

1. In the designer, double-click the component to which you wish to apply logic.
   The **Code Editor** appears, cursor in place between the reserved words begin and end in the event handler.
2. Code your application logic.
3. Save and compile the application.

Building an Application with XML Components

This example creates a VCL Forms application that uses an XMLDocument component to display contents in an XML file.

The basic steps are:

1. Create an XML document.
2. Create a VCL form.
3. Place an XMLDocument component on the form, and associate it with the XML file.
4. Create VCL components to enable the display of XML file contents.
5. Write event handlers to display XML child node contents.
6. Compile and run the application.

To create the XML document

1. Copy the text below into a file in a text editor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings [
    <!ELEMENT StockHoldings (Stock+)>
    <!ELEMENT Stock (name)>
    <!ELEMENT Stock (price)>
    <!ELEMENT Stock (symbol)>
    <!ELEMENT Stock (shares)>
]>

<StockHoldings>
  <Stock exchange="NASDAQ">
    <name>Borland</name>
    <price>10.375</price>
    <symbol>BORL</symbol>
    <shares>100</shares>
  </Stock>

  <Stock exchange="NYSE">
    <name>MyCompany</name>
    <price>8.75</price>
    <symbol>MYCO</symbol>
    <shares type="preferred">25</shares>
  </Stock>
</StockHoldings>
```

2. Save the file to your local drive as an XML document. Give it a name such as stock.xml.
3. Open the document in your browser.

   The contents should display without error.

> **Note:** In the browser, you can choose **View** ▸ **Source** to view the source file in the text editor.

To create a form with an XMLDocument component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Internet** page on the **Tool Palette**, place an TXMLDocument component on the form.
3. In the **Object Inspector**, click the ellipsis button next to the FileName property, browse to the location of the XML file you created, and open it.

   The XML file is associated with the TXMLDocument component.
4. In the **Object Inspector**, set the Active property to true.

To set up the VCL components

1. From the **Standard** page on the **Tool Palette**, place a TMemo component on the form.
2. From the **Standard** page on the **Tool Palette**, place two TButton components on the form just above **Memo1**.
3. In the **Object Inspector** with **Button1** selected, enter Borland for the Caption property.
4. In the **Object Inspector** with **Button2** selected, enter MyCompany for the Caption property.

To display child node contents in the XML file

1. In the **Object Inspector** with **Button1** selected, double-click the OnClick event on the **Events** tab.

   The Code displays with the cursor in the `TForm1.Button1Click` event handler block.
2. Enter the following code to display the stock price for the first child node when the Borland button is clicked:

```
[Delphi]
  BorlandStock:=XMLDocument1.DocumentElement.ChildNodes[0];
  Price:= BorlandStock.ChildNodes['price'].Text;
  Memo1.Text := Price;
```

```
[C++]
IXMLNode *BorlandStock = XMLDocument1->DocumentElement->
            ChildNodes->GetNode(0);
WideString Price = BorlandStock->ChildNodes->
            FindNode( "price" )->Text;
Memo1->Text = Price;
```

3. For Delphi, add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var
     BorlandStock: IXMLNode;
  Price: string;
```

4. In the **Object Inspector** with **Button2** selected, double-click the OnClick event on the **Events** tab.

   The Code displays with the cursor in the `TForm1.Button2Click` event handler block.
5. Enter the following code to display the stock price for the second child node when the MyCompany button is clicked:

```
[Delphi]

  MyCompany:=XMLDocument1.DocumentElement.ChildNodes[1];
  Price:= MyCompany.ChildNodes['price'].Text;
  Memo1.Text := Price;
```

```
[C++]
IXMLNode *MyCompany = XMLDocument1->DocumentElement
        ->ChildNodes->GetNode(1);
WideString Price = BorlandStock->ChildNodes
        ->FindNode( "price" )->Text;
Memo1->Text = Price;
```

6 For Delphi, add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var
  MyCompany: IXMLNode;
  Price: string;
```

To compile and run the application

1 Choose **Run** ▸ **Run** to compile and execute the application.

The application form displays two buttons and a memo.

2 Click the **Borland** button.

The stock price displays.

3 Click the **MyCompany** button.

The stock price displays.

Building Application Menus

Menus provide an easy way for your users to execute logically grouped commands. You can add or delete menu items, or drag them to rearrange them during designtime. In addition to **TMainMenu** and **TPopupMenu** components, the **Tool Palette** also contains **TActionMainMenuBar**, **TActionManager**, and **TActionToolBar**.

To create application menus

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **Application**.

   The **Windows Designer** displays.
3. Build application menus.
4. Use the Menu Designer.
5. Create an event handler for each menu item.
6. Move menu items.
7. Add images to menu items.

To build application menus

1. From the **Standard** category of the **Tool Palette**, add **TMainMenu** or **TPopupMenu** component to your form.

   A visual representation of the menu appears on the designer.

   Note: A **TMainMenu** component creates a menu that is attached to the title bar of the form. A **TPopupMenu** component creates a menu that appears when the user right-clicks in the form.
2. To view the menu, if the form is visible, click the form.

   The menu appears in the form exactly as it will when you run the program.
3. To delete a menu item, select the menu item you want to delete. Press Delete.
4. To edit menu items, select the Windows form, select the menu item you want to edit, and edit its properties.
5. To make the menu item a separator bar, place the cursor on the menu where you want a separator to appear and enter a hyphen (-) for the caption or press the hyphen (-) key.
6. To specify a keyboard shortcut for a menu item, in the **Object Inspector**, set the ShortCut property.

To use the Menu Designer

1. Select a menu component on the form.
2. Double-click the menu component.

   The **Menu Designer** window opens.

   Note: You can also open the **Menu Designer** by clicking the ellipsis(...) button next to the Items property in the **Object Inspector**.
3. To name a menu component, in the **Object Inspector**, set the Caption property.

Tip: Delphi derives the Name property from the caption, for e.g. if you give a menu item a Caption property value of File, Delphi assigns the menu item a Name property of File1. However, if you fill in the Name property before filling in the Caption property, Delphi leaves the Caption property blank until you type a value.

4 Right-click anywhere on the Menu Designer to use the **Menu Designer** context menu.

   A drop-down list opens. This is the **Menu Designer** context menu.

5 To insert a placeholder below or to the right of the cursor, choose **Insert** from the context menu.

6 To delete the selected menu item (and all its subitems, if any), click **Delete** from the context menu.

7 To switch among menus in a form, choose **Select Menu** from the context menu.

   The **Select Menu** dialog box appears. It lists all the menus associated with the form whose menu is currently open in the Menu designer.

8 From the list in the **Select Menu** dialog box, choose the menu you want to view or edit.

To create an event handler for a menu item

1 In the designer, double-click the menu item to which you wish to add an event handler.

   The **Code Designer** appears, cursor in place between event handler brackets.

2 Code your menu item logic.

3 Save and compile the application.

To move menu items

1 To move a menu item along the desired menu bar, drag the item until the arrow tip of the cursor points to the new location.

2 Release the mouse button.

3 To move a menu item into a menu list, drag the item until the arrow tip of the cursor points to the new menu.

4 Release the mouse button.

To add images to menu items

1 From the **Tool Palette**, drag a **TMainMenu** or **TPopupMenu** component to a form.

2 From the **Tool Palette**, drop a **TImageList** component to the form.

3 Double-click on the **TImageList** component.

   The **ImageList** editor opens.

4 Click **Add** to select the bitmap or bitmap group you want to use in the menu.

5 Select the bitmap that you want and click **OK**.

6 In the **Object Inspector**, set the **Images** property of the **TMainMenu** or **TPopupMenu** component to the image you selected in the **ImageList** editor.

Building VCL Forms Applications With Graphics

Each of the procedures listed below builds a VCL Form application that uses graphics. Build one or more of the examples and then add other graphics features to these basic VCL Form applications.

1. Draw straight lines.
2. Draw rectangles and ellipses.
3. Draw a polygon.
4. Display a bitmap image.
5. Place a bitmap in a combo box.

Copying a Complete String List

Copying a string list can have the effect of appending to or overwriting an existing string list. This VCL application appends to a string list. With a simple change, it can overwrite a string list. Creating this VCL application consists of the following steps:

1. Create a VCL Form with TButtons, TComboBox, and TMemo controls.
2. Write the code to create a string list to the Button1 OnClick handler.
3. Write the code to copy the string list to the Button2 OnClick handler.
4. Run the application.

To create a VCL Form with Button, ComboBox, and Memo controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons, a TComboBox, and a TMemo component on the form.

To create the string list

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press return.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declarations:

   ```
   [Delphi]
   var
     StringList: TStrings;
   ```

   For C++, enter the following variable declarations:

   ```
   [C++]
   TStrings *StringList;
   ```
5. Insert the cursor within the code block, and type the following code:

   ```
   [Delphi]
   StringList := TStringList.Create;
     try
       with StringList do begin
         Add('This example uses a string List.');
         Add('It is the easiest way to add strings');
         Add('to a combobox''s list of strings.');
         Add('Always remember: the TStrings.Create');
   ```

```
      Add('method is abstract; use the');
      Add('TStringList.Create method instead.');
    end;

    with ComboBox1 do begin
                        Width := 210;
      Items.Assign(StringList);
      ItemIndex := 0;
    end;
  finally
    StringList.free;
  end;
```

```
[C++]
StringList = new TStringList();
try {
  StringList->Add( "This example uses a string list" );
  StringList->Add( "It is the easiest way to add strings" );
  StringList->Add( "to a ComboBox's list of strings." );
  StringList->Add( "Remember to call the TStringList constructor!" );
  ComboBox1->Width = 210;
  ComboBox1->Items->Assign( StringList );
  ComboBox1->ItemIndex = 0;
} __finally {
  StringList->Free();
}
```

To copy the string list

1. Select Button2 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.Button2Click` (Delphi) or `TForm1::Button2Click` (C++) event handler block. At the cursor, enter the following code:

```
[Delphi]
Memo1.Lines.AddStrings(ComboBox1.Items);
```

```
[C++]
Memo1->Lines->AddStrings( ComboBox1->Items
```

To run the application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application.

   The form displays with the controls.
2. Click Button1.
3. In ComboBox1, click the arrow to expand the drop-down list.

   The strings display in the TComboBox in the order listed in the event handler code for Button1.
4. Click Button2.

   In the Memo1 window, the strings from ComboBox1 are appended to the 'Memo1' string.

Note: Try replacing the code in the Button2 event handler with the following code; then compile and run the application again.

```
[Delphi]
Memo1.Lines.Assign(ComboBox1.Items);
```

```
[C++]
Memo1->Lines->Assign( ComboBox1->Items );
```

The strings from ComboBox1 overwrite the 'Memo1' string.

Copying Data From One Stream To Another

Creating this VCL application consists of the following steps:

1. Create a project directory containing a text file to copy.
2. Create a VCL Form with a button control.
3. Write the code to read the string and write it to a file.
4. Run the application.

To set up your project directory and a text file to copy

1. Create a directory in which to store your project files.
2. Using a text editor, create a simple text file and save it as from.txt in your project directory.

To create a VCL Form with a button control

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton component on the form.
3. In the **Object Inspector**, enter CopyFile for the Caption and Name properties.

To write the copy stream procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.CopyFileClick` (Delphi) or `TForm1::CopyFileClick` (C++) event handler block.
3. For Delphi, Place the cursor before the `begin` reserved word; then press return.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

   ```
   var stream1, stream2: TStream;
   ```

   For C++, enter the following variable declarations:

   ```
   [C++]
   TStream *stream1, *stream2;
   ```

5. Insert the cursor within the code block, and type the following code:

   ```
   [Delphi]
   stream1 := TFileStream.Create('from.txt', fmOpenRead);
   try
     stream2:= TFileStream.Create('to.txt', fmCreate);
     try
   ```

```
    stream2.CopyFrom(stream1, stream1.Size);
  finally
    stream2.Free;
  end;
finally
  stream1.Free;
end;
```

```
[C++]
stream1 = new TFileStream( "from.txt", fmOpenRead );
try {
  stream2 = new TFileStream( "to.txt", fmCreate );
      try {
    stream2->CopyFrom( stream1, stream1->Size );
  } __finally {
    stream2->Free();
  }
} finally {
  stream1->Free();
}
```

To run the application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application.
   The form displays with a button called **CopyFile**.
2. Click **CopyFile**.
3. Use a text editor to open the newly created file to.txt, which is located in your project directory.
   The contents of from.txt are copied into to.txt.

Creating a New VCL Component

You can use the **New VCL Component** wizard to create a new VCL component. The wizard detects which personality of the product you are using and creates the appropriate type of component.

To create a new VCL component

1. Specify an ancestor component.
2. Specify the class name.
3. Create a unit or add the unit to a package.

To specify an ancestor component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. Choose **Component** ▸ **New VCL Component**.

   This displays the first page of the **New VCL Component** wizard. By default, it should be set to **Delphi for VCL Win32**.
3. Click **Next**.

   This displays the second page of the **New VCL Component** wizard and loads the page with ancestor components.
4. Select an ancestor component from the list.
5. Click **Next**.

   This displays the third page of the **New VCL Component** wizard.

To specify a class name

1. If you want to change the default class name, enter it in the **Class Name** textbox.
2. Enter the name of the area on the **Tool Palette** where you want the component to appear in the **Palette Page** textbox.
3. Enter the unit name in the **Unit Name** textbox.
4. Enter the search path in the **Search Path** textbox.
5. Click **Next**.

> **Note:** You can also take the default values.

To create a unit

1. Select the **Create Unit** radio button.
2. Click **Finish**.

To install a unit into an existing package

1. Select the **Install into Existing Package** radio button.

2 Click **Next**.

This generates a list of existing packages.

3 Select the package you want to install the unit into.

4 Click **Finish**.

To install a unit into a new package

1 Select the **Install into New Package** radio button.

2 Click **Next**.

3 Enter a name for the package into the **File Name** textbox.

4 Enter a description for the package into the **Description** textbox.

5 Click **Finish**.

The new unit opens in the **Code Editor**.

Creating a VCL Form Instance Using a Local Variable

A safe way to create a unique instance of a modal VCL form is to use a local variable in an event handler as a reference to a new instance. If you use a local variable, it doesn't matter whether the form is auto-created or not. The code in the event handler makes no reference to the global form variable. Using Developer Studio 2006, the following procedure creates a modal form instance dynamically. It (optionally) removes the second form's invocation at startup.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup (optional).
4. Link the forms.
5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **C++Builder Projects** and double-click the **Form** icon.

   The **VCL Forms Designer** displays Form2.

To optionally remove Form2's invocation at startup

1. Choose **Project** ▸ **Options** ▸ **Forms**.

   The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**.

   Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. Select Form1 and choose **File** ▸ **Use Unit** (Delphi) or **Include Unit Hdr** (C++).

   The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

For Delphi, a uses clause containing the unit name `Unit2` is placed in the implementation section of Unit1.

For C++, the `#include "Unit2.h"` directive is added to Unit1.h.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a TButton on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, insert the cursor just above the event handler block and enter the following statement to define a local variable:

   ```
   [Delphi]
   var FM: TForm2;
   ```

   For C++, enter the following variable declaration:

   ```
   [C++]
   TForm2 *FM;
   ```
4. Insert the cursor in the event handler block, and enter the following code:

   ```
   [Delphi]
   FM := TForm2.Create(self);
   FM.ShowModal;
   FM.Free;
   ```

   ```
   [C++]
   FM = new TForm2( this );
   FM->ShowModal();
   FM->Free();
   ```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**.

   The application executes, displaying Form1.
2. Click the button.

   Form2 displays.
3. With Form2 displayed, attempt to click on Form1 to activate it.

   Nothing happens. Click the X in the upper right corner of Form2.

   Form2 closes and Form1 becomes the active form.

Creating a VCL Forms ActiveX Active Form

Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder, for example.

This procedure uses the VCL forms ActiveX Active Form wizard to create an Active Form containing two components. To test the control, you can deploy it to the Web. This procedure consists of the following major steps:

1. Create an ActiveX library project for an ActiveX Active Form.
2. Add controls to the Active Form.
3. Add event handling code for the controls.
4. Deploy the project to the Web.
5. Display the form and test the controls in your Web browser.

To create an Active X library project for an ActiveX Active Form

1. Create a directory on your local drive for the ActiveX project. Give it an easy to find name, for example, ActiveX.
2. Create a second directory to contain the ActiveX component and an HTML file for deploying the Active Form to your Microsoft Internet Explorer Web browser. Name this directory ActiveX_Deploy.
3. Choose **File** ▸ **New** ▸ **Other** and select the **ActiveX** page in the **New Items** dialog.
4. On the **ActiveX** page, double-click Active Form.

   The **Active Form Wizard** displays.
5. Accept the default settings and click **OK**.

   The wizard generates the code needed to implement the ActiveX control and adds the code to the project. If the project is already an ActiveX library, the wizard adds the control to the current project.

   Note: If the project is not already an ActiveX library, a **Warning** dialog displays and asks you if you want to start a new ActiveX library project.
6. Click **OK** to start a new ActiveX library project.

   An ActiveX Active Form displays.

To add some functionality to the Active Form

1. From the **Standard** page of the **Tool Palette**, add TEdit and TButton components to the form.
2. Select the button.
3. On the **Events** tab in the **Object Inspector**, double-click the OnClick event.

   The **Code Editor** opens with the cursor in place in the `TActiveFormX.Button1Click` (Delphi) or `TActiveFormX::Button1Click()` (C++) event handler block.

   Enter the following code at the cursor:

```
[Delphi]
ShowMessage(Edit1.text);
```

```
[C++]
ShowMessage( Edit1->Text )
```

4 Save the project files to your ActiveX directory.

To deploy the Active Form to your Web browser

1 Choose **Project** ▸ **Web Deployment Options...**.
The **Web Deployment Options** dialog displays.
2 On the **Project** page, use the **Browse** button to enter the path to the ActiveX_Deploy directory.
3 Enter the same path for the **HTML dir**.
4 For **Target URL**, enter **.** to indicate the current directory.
5 Click **OK**.
6 Choose **Project** ▸ **Web Deploy**.
HTML and OCX files are created in the ActiveX_Deploy directory.

To test the Active Form

1 Launch your browser.
2 Choose **File** ▸ **Open**, and browse to the ActiveX_Deploy directory.
3 Double-click the HTML file to launch it in the browser window.
The Active Form displays in the browser window.
4 Click the button.
A pop-up dialog displays the text in the Edit box.
5 Change the text, and click the button again.
The new text you entered displays in the pop-up.

Creating Actions in a VCL Forms Application

Using Developer Studio 2006, the following procedure illustrates how to create actions using the ActionList tool. It sets up a simple application and describes how to create a file menu item with a file open action.

Building the VCL application with ActionList actions consists of the following major steps:

1. Create a main window and add tools for creating a main menu and a File open action.
2. Add the File category to the main menu.
3. Add the File open action to the File category.
4. Build and run the application.

To create a main window

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays.
2. From the **Standard** category of the **Tool Palette**, add a TMainMenu and TActionList component to the form.

   Tip: To display icons for nonvisual components such as ActionList1, choose **Tools** ▸ **Options** ▸ **Environment Options**, select VCL Designer from the Delphi options, click Show Component Options, and click **OK**.

To add the File category to the main menu

1. Double-click MainMenu1 on the form.

   The MainMenu1 editor displays with the first blank command category selected.
2. In the **Object Inspector**, enter &File for the Caption property and press RETURN.

   File displays on the main menu.
3. Click File on the MainMenu1 editor.

   The first blank action under the File command displays. Select the blank action.
4. Double-click ActionList1 on the form.

   The ActionList editor displays.
5. In the editor, select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog.
6. Scroll to the File category, and click the TFileOpen action.
7. Click **OK** to close the dialog.

   File displays in the Categories listbox in the ActionList editor.
8. Click File in the editor.

   The FileOpen1 action displays in the Action listbox.

To add the File Open action to the File category

1. Double-click MainMenu1, if necessary, to display the MainMenu1 editor; select the blank action under the File category.

2 In the **Object Inspector**, enter &Open for the Caption property and select FileOpen1 from the Action property drop-down list; then press RETURN.

   Open... displays in the blank action field in the MainMenu1 editor.

To build and run the application

1 Choose **Run** ▸ **Run**.

   The application executes, displaying a form with the main menu bar and the File menu.

2 Choose **File** ▸ **Open** in the application.

   The standard **Open** file dialog displays.

Creating Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with TButton and TComboBox controls.
2. Write the code to create strings to the TButton OnClick handler.
3. Run the application.

To create a VCL Form with TButton and TComboBox controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton, a TLabel, and a TComboBox component on the form.

To write the create string procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press RETURN.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declarations:

```
var
  StringList: TStrings;
```

   For C++, enter the following variable declarations:

```
[C++]
TStrings *StringList;
```

5. Insert the cursor within the code block, and type the following code:

```
[Delphi]
StringList := TStringList.Create;
  try
    with StringList do begin
      Add('Animals');
      Add('Cats');
      Add('Flowers');
    end;

    with ComboBox1 do begin
                        Width := 210;
      Items.Assign(StringList);
      ItemIndex := 0;
```

```
    end;

    Label1.Caption := 'Flowers has an index of ' +
      IntToStr( StringList->IndexOf( 'Flowers' ) );
  finally
    StringList.free;
  end;
```

```
[C++]
StringList = new TStringList();
try {
  StringList->Add( "Animals" );
  StringList->Add( "Cats" );
  StringList->Add( "Flowers" );
  ComboBox1->Width = 210;
  ComboBox1->Items->Assign( StringList );
  ComboBox1->ItemIndex = 0;
  Label1->Caption = "Flowers has an index of " +
    IntToStr( StringList->IndexOf( "Flowers" ) );
} __finally {
      StringList->Free();
}
```

To run the application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application.

   The form displays with the controls.

2. Click the Button.

   The strings 'Animals', 'Cars', and 'Flowers' display alphabetically in a list in the ListBox. The Label caption displays the message string: 'Flowers has an index value of 2.'

3. In the ComboBox, click the arrow to expand the drop-down list.

   The strings added in the TButton event handler appear.

Defining the Thread Object

To define the thread object

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Thread Object** icon.

   The **New Thread Object** dialog displays.
2. Enter a class name, for example, TMyThread.
3. Optionally check the **Named Thread** check box, and enter a name for the thread, for example, MyThreadName.

   Tip: Entering a name for Named Thread makes it easier to track the thread while debugging.
4. Click **OK**.

The **Code Editor** displays the skeleton code for the thread object.

The code generated for the new unit will look like this if you named your thread class TMyThread.

```
[Delphi]
unit Unit1;

interface

uses
  Classes;

type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in visual components can only be
  used in a method called using Synchronize, for example,

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{ TMyThread }

procedure TMyThread.Execute;
begin
  { Place thread code here }
end;

end.
```

Adding a name for the thread generates additional code for the unit. It includes the Windows unit, adds the procedure (Delphi) or function (C++) `SetName`, and adds the record `TThreadNameInfo` (Delphi) or struct `THREADNAME_INFO` (C++). The name is assigned to the `FName` field, as shown here:

```
[Delphi]
unit Unit1;

interface

uses
  Classes {$IFDEF MSWINDOWS} , Windows {$ENDIF};

type
  TMyThread = class(TThread)
  private
    procedure SetName;
  protected
    procedure Execute; override;
  end;
```

```
implementation

{ Important: Methods and properties of objects in visual components can only be
  used in a method called using Synchronize, for example,

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{$IFDEF MSWINDOWS}
type
  TThreadNameInfo = record
    FType: LongWord;     // must be 0x1000
    FName: PChar;        // pointer to name (in user address space)
    FThreadID: LongWord; // thread ID (-1 indicates caller thread)
    FFlags: LongWord;    // reserved for future use, must be zero
  end;
{$ENDIF}

{ TMyThread }

procedure TMyThread.SetName;
{$IFDEF MSWINDOWS}
var
  ThreadNameInfo: TThreadNameInfo;
{$ENDIF}
begin
{$IFDEF MSWINDOWS}
  ThreadNameInfo.FType := $1000;
  ThreadNameInfo.FName := 'MyThreadName';
  ThreadNameInfo.FThreadID := $FFFFFFFF;
  ThreadNameInfo.FFlags := 0;

  try
    RaiseException( $406D1388, 0, sizeof(ThreadNameInfo) div sizeof(LongWord),
@ThreadNameInfo );
  except
  end;
{$ENDIF}
end;

procedure TMyThread.Execute;
begin
  SetName;
  { Place thread code here }
end;

end.
```

```
[C++]
// Unit1.h

#ifndef Unit1H
#define Unit1H
#include <Classes.hpp>
```

```
class TMyThread : public TThread
{
  typedef struct tagTHREADNAME_INFO
  {
    DWORD dwType;
    LPCSTR szName;
    DWORD dwThreadID;
    DWORD dwFlags;
  } THREADNAME_INFO;
private:
  void SetName();
protected:
  void __fastcall Execute();
public:
  __fastcall TMyThread(bool CreateSuspended);
};

#endif
```

```
[C++]
// Unit1.cpp
#include "Unit3.h"
#pragma package(smart_init)

__fastcall TMyThread::TMyThread(bool CreateSuspended)
  : TThread(CreateSuspended)
{
}

void TMyThread::SetName()
{
  THREADNAME_INFO info;
  info.dwType = 0x1000;
  info.szName = "TMyThreadName";
  info.dwThreadID = -1;
  info.dwFlags = 0;

  __try
  {
     RaiseException( 0x406D1388, 0, sizeof(info)/sizeof(DWORD),(DWORD*)&info; );
  }
  __except (EXCEPTION_CONTINUE_EXECUTION)
  {
  }
}

void __fastcall TMyThread::Execute()
{
  SetName();
  //---- Place thread code here ----
}
```

Deleting Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with Buttons and ListBox controls.
2. Write the code to add strings to a list.
3. Write the code to delete a string from the list.
4. Run the application.

To create a VCL Form with TButton and ListBox controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons and a TListBox component on the form.
3. Select Button1 on the form.
4. In the **Object Inspector**, enter Add for the Name and Caption properties.
5. Select Button2 on the form.
6. In the **Object Inspector**, enter Delete for the Name and Caption properties.

To add strings to a list

1. Select the **Add** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.AddClick` (Delphi) or `TForm1::AddClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press return.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

   ```
   var
     MyList: TStringList;
   ```

   For C++, enter the following variable declaration:

   ```
   [C++]
   TStringList *MyList;
   ```

5. Insert the cursor within the code block, and type the following code:

   ```
   [Delphi]
   MyList := TStringList.Create;
       try
         with MyList do
         begin
           Add('Mice');
           Add('Goats');
   ```

```
        Add('Elephants');
        Add('Birds');
        ListBox1.Items.AddStrings(MyList);
      end;
    finally
      MyList.Free;
    end;
```

```
[C++]
MyList = new TStringList();
try {
  MyList->Add( "Mice" );
  MyList->Add( "Goats" );
  MyList->Add( "Elephants" );
  MyList->Add( "Birds" );
  ListBox1->Items->AddStrings( MyList );
} __finally {
  MyList->Free();
}
```

To delete a string from the list

1. Select the **Delete** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.DeleteClick` (Delphi) or `TForm1::DeleteClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press ENTER.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

   ```
   [Delphi]
   var
     BIndex: Integer;
   ```

   For C++, enter the following variable declaration:

   ```
   [C++]
   int BIndex;
   ```
5. For Delphi, insert the cursor within the code block and type the following code:

   ```
   [Delphi]
   with ListBox1.Items do
       begin
         BIndex := IndexOf('Elephants');
         Delete (BIndex);
       end;
   ```

```
[C++]
BIndex = ListBox1->Items->IndexOf( "Elephants" );
ListBox1->Items->Delete( BIndex );
```

To run the application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application.
   The form displays with the controls.
2. Click the **Add** button.
   The strings 'Mice', 'Goats', 'Elephants', and 'Birds' display in the order listed.
3. Click the **Delete** button.
   The string 'Elephants' is deleted from the list.

Displaying a Bitmap Image in a VCL Forms Application

This procedure loads a bitmap image from a file and displays it to a VCL form.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's onClick event handler to load and display a bitmap image.
4. Build and run the application.

To create a VCL form and button

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the **Tool Palette**, place a TButton component on the form.

To provide a bitmap image

1. Create a directory in which to store your project files.
2. Locate a bitmap image on your local drive, and copy it to your project directory.
3. Save all files in your project to your project directory.

To write the OnClick event handler

1. In the **Object Inspector**, double-click the Button1 OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with path to the bitmap image in your project directory:

```
[Delphi]
Bitmap := TBitmap.Create;
try
  Bitmap.LoadFromFile('MyFile.bmp');
  Form1.Canvas.Brush.Bitmap := Bitmap;
  Form1.Canvas.FillRect(Rect(0,0,100,100));
finally
  Form1.Canvas.Brush.Bitmap := nil;
  Bitmap.Free;
end;
```

```
[C++]
Graphics::TBitmap *Bitmap = new Graphics::TBitmap();
try {
  Bitmap->LoadFromFile( "..\\MyFile.bmp" );
  Form1->Canvas->Brush->Bitmap = Bitmap;
  Form1->Canvas->FillRect( Rect(0,0,100,100) );
} __finally {
```

```
  Form1->Canvas->Brush->Bitmap = NULL;
  Bitmap->Free();
}
```

Note: For C++ projects, the code assumes the target output directory is located in the project directory.

Tip: You can change the size of the rectangle to be displayed by adjusting the `Rect` parameter values.

3 For Delphi, add the following variable declaration in the var block:

```
Bitmap : TBitmap;
```

To run the program

1 Select **Run** ▸ **Run**.

2 Click the button to display the image bitmap in a 100 x 100-pixel rectangle in the upper left corner of the form.

Displaying a Full View Bitmap Image in a VCL Forms Application

This procedure loads a bitmap image from a file and displays it in its entirety to a VCL form. The procedure uses the Height and Width properties of the Bitmap object to display a full view of the image.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's onClick event handler to load and display a bitmap image.
4. Build and run the application.

To create a VCL form and button

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the **Tool Palette**, place a button component on the form.
3. In the **Object Inspector**, enter Full View for the Caption property and FullView for the name property.

To provide a bitmap image

1. Create a directory in which to store your project files.
2. Locate a bitmap image on your local drive, and copy it to your project directory.
3. Save all files in your project to your project directory.

To write the OnClick event handler

1. In the **Object Inspector**, double-click the Button1 OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.FullViewClick` (Delphi) or `TForm1::FullViewClick` (C++) event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with the name of the bitmap image in your project directory:

```
[Delphi]

      Bitmap := TBitmap.Create;
      try
        Bitmap.LoadFromFile('MyFile.bmp');
        Form1.Canvas.Brush.Bitmap := Bitmap;
        Form1.Canvas.FillRect(Rect(0,0,Bitmap.Width,Bitmap.Height));
      finally
        Form1.Canvas.Brush.Bitmap := nil;
        Bitmap.Free;
      end;
```

```
[C++]
Graphics::TBitmap Bitmap = new Graphics::TBitmap();
try {
  Bitmap->LoadFromFile( "..\\MyFile.bmp" );
```

```
  Form1->Canvas->Brush->Bitmap = Bitmap;
  Form1->Canvas->FillRect(
    Rect( 0, 0, Bitmap->Width, Bitmap->Height ) );
} __finally {
  Form1->Canvas->Brush->Bitmap = NULL;
  Bitmap->Free();
}
```

Note: For C++ projects, the code assumes the target output directory is located in the project directory.

3 For Delphi, add the following variable declaration in the var block:

```
[Delphi]
Bitmap : TBitmap;s
```

To run the program

1 Choose **Run** ▸ **Run**.

2 Click the button to display the image bitmap in a rectangle in the upper left corner of the form.

Displaying an Auto-Created VCL Form

Using Developer Studio 2006, the following procedure creates a modal form at design time that is displayed later during program execution.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Link the forms.
4. Create a control on the main form to display the modal form; then write the event handler.
5. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays Form1.

2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Form** icon.

   The **VCL Forms Designer** displays Form2.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**.

   For C++, select Form1 and choose **File** ▸ **Include Unit Hdr**.

   The **Uses Unit** dialog displays.

2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

For Delphi, a uses clause containing the unit name Unit2 is placed in the implementation section of Unit1.

For C++, the `#include "Unit2.h"` directive is added to Unit1.h.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a button on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.

3. Enter the following event handling code:

```
[Delphi]
Form2.ShowModal;
```

```
[C++]
Form2->ShowModal();
```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**.
   The application executes, displaying Form1.
2. Click the button.
   Form2 displays.
3. Click the **X** in the upper right corner of Form2.
   Form2 closes and Form1 becomes the active form.

Drawing a Polygon in a VCL Forms Application

This procedure draws a polygon in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a polygon.
3. Build and run the application.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, click the **Events** tab.
2. Double-click the OnPaint event.

   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
3. Enter the following event handling code:

```
[Delphi]
Canvas.Polygon ([Point(0,0), Point(0, ClientHeight),
Point(ClientWidth, ClientHeight)]);
```

```
[C++]
TPoint points[] = { Point(0,0),
                    Point(0, ClientHeight),
                    Point(ClientWidth, ClientHeight) };
Canvas->Polygon( points, 3 );
```

To run the program

1. Select **Run** ▸ **Run**.
2. The applications executes, displaying a right triangle in the lower left half of the form.

Drawing a Rounded Rectangle in a VCL Forms Application

This procedure draws a rounded rectangle in a VCL form.

1. Create a VCL form and code the form's OnPaint event handler.
2. Build and run the application.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. In the **Object Inspector**, click the **Events** tab.
3. Double-click the OnPaint event.

   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
4. Enter the following event handling code:

```
[Delphi]
Canvas.RoundRect(0, 0, ClientWidth div 2,
  ClientHeight div 2, 10, 10);
```

```
[C++]
Canvas->RoundRect( 0, 0, ClientWidth / 2, ClientHeight / 2, 10, 10 );
```

To run the program

1. Save all files in your project; then choose **Run** ▸ **Run**.
2. The application executes, displaying a rounded rectangle in the upper left quadrant of the form.

Drawing Rectangles and Ellipses in a VCL Forms Application

This procedure draws a rectangle and ellipse in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a rectangle and ellipse.
3. Build and run the application.

To create a VCL form and place an image on it

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, double-click the Form1 OnPaint event on the Events tab.

   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
2. Enter the following event handling code:

```
[Delphi]
Canvas.Rectangle (0, 0, ClientWidth div 2, ClientHeight div 2);
Canvas.Ellipse (0, 0, ClientWidth div 2, ClientHeight div 2);
```

```
[C++]
Canvas->Rectangle( 0, 0, ClientWidth / 2, ClientHeight / 2 );
Canvas->Ellipse( 0, 0, ClientWidth / 2, ClientHeight / 2 );
```

To run the program

1. Choose **Run** ▸ **Run**.
2. The applications executes, displaying a rectangle in the upper left quadrant, and an ellipse in the same area of the form.

Drawing Straight Lines In a VCL Forms Application

This procedure draws two diagonal straight lines on an image in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw the straight lines.
3. Build and run the application.

To create a VCL form and place an image on it

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, double-click the Form1 OnPaint event on the **Events** tab.

   The **Code Editor** displays with the cursor in the `TForm1.FormPaint` event handler block.
2. Enter the following event handling code:

```
[Delphi]
with Canvas do
begin
MoveTo(0,0);
LineTo(ClientWidth, ClientHeight);
MoveTo(0, ClientHeight);
LineTo(ClientWidth, 0);
end;
```

```
[C++]
Canvas->MoveTo( 0, 0 );
Canvas->LineTo( ClientWidth, ClientHeight );
Canvas->MoveTo( 0, ClientHeight );
Canvas->LineTo( ClientWidth, 0 );
```

To run the program

1. Choose **Run** ▸ **Run**.
2. The applications executes, displaying a form with two diagonal crossing lines.

Tip: To change the color of the pen to green, insert this statement following the first `MoveTo()` statement in the event handler code: `Pen.Color := clGreen;` (Delphi) `Canvas->Pen->Color = clGreen;` (C++). Experiment using other canvas and pen object properties. See "Using the properties of the Canvas object" in the Delphi 7 Developer's Guide.

Dynamically Creating a VCL Modal Form

You may not want all your VCL application's forms in memory at once. To reduce the amount of memory required at load time, your application can create forms only when it needs to make them available for use. A dialog box, for example, needs to be in memory only during the time the user interacts with it. Using Developer Studio 2006, the following procedure creates a modal form dynamically. The main difference between dynamically creating a form and displaying an auto-created VCL form is that you remove the second form's invocation at startup and write code to dynamically create the form.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup.
4. Link the forms.
5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Form** icon.

   The **VCL Forms Designer** displays Form2.

To remove Form2's invocation at startup

1. Choose **Project** ▸ **Options** ▸ **Forms**.

   The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**.

   Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**.

   For C++, select Form1 and choose **File** ▸ **Include Unit Hdr**

   The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

   For Delphi, a uses clause containing the unit name Unit2 is placed in the implementation section of Unit1.

For C++, the `#include "Unit2.h"` directive is added to Unit1.h.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a TButton on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab.
   The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. Enter the following event handling code:

```
Form2 := TForm2.Create(self);
try
  Form2.ShowModal;
finally
  Form2.Free;
end;
```

```
Form2 = new TForm2( this );
try {
  Form2->ShowModal();
} __finally {
  Form2->Free();
}
```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**.
   The application executes, displaying Form1.
2. Click the button.
   Form2 displays.
3. Click the **X** in the upper right corner of the form.
   Form2 closes and Form1 becomes the active form.

Dynamically Creating a VCL Modeless Form

A modless form is a window that is displayed until it is either obscured by another window or until it is closed or minimuzed by the user. Using Developer Studio 2006, the following procedure creates a modeless form dynamically.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup.
4. Link the forms.
5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Form** icon.

   The **VCL Forms Designer** displays Form2.

To remove Form2's invocation at startup

1. Choose **Project** ▸ **Options**.

   The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**.

   Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**.

   For C++, select Form1 and choose **File** ▸ **Include Unit Hdr**.

   The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

   For Delphi, a uses clause containing the unit name Unit2 is placed in the implementation section of Unit1.

   For C++, the `#include "Unit2.h"` directive is added to Unit1.h.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a button on the form.

2 In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab.

The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++)event handler block.

3 Enter the following event handling code:

```
[Delphi]
Form2 := TForm2.Create(self);
Form2.Show;
```

```
[C++]
Form2 = new TForm2( this );
Form2->Show();
```

Note: If your application requires additional instances of the modeless form, declare a separate global variable for each instance. In most cases you use the global reference that was created when you made the form (the variable name that matches the Name property of the form).

To build and run the application

1 Save all files in the project; then choose **Run** ▸ **Run**.

The application executes, displaying Form1.

2 Click the button.

Form2 displays.

3 Click Form1.

Form1 becomes the active form. Form2 displays until you minimize or close it.

Handling Exceptions

To handle exceptions in the thread function

1. Add a try...except block to the implementation of your `Execute` method.
2. Code the logic such as shown here:

```
[Delphi]
procedure TMyThreadExecute;
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    {do something with exceptions}
  end;
end;
```

```
[C++]
void __fastcall TMyThread::Execute() {
  try {
    while( !Terminated() ) {
      // perform tasks
    }
  } catch(...) { // catch specific exceptions first
    // exception-handling code
  }
}
```

Initializing a Thread

To initialize a thread object

1 Assign a default thread priority.
2 Indicate when the thread is freed.

To assign a default priority

1 Assign a default priority to the thread from the values listed in the table below.

Use a high-priority to handle time critical tasks, and a low priority to perform other tasks.

Value	Priority
tpIdle	The thread executes only when the system is idle. Windows won't interrupt the other threads to execute a thread with tpIdle priority.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.
tpHighest	The thread's priority is two points above normal.
tpTimeCritical	The thread gets highest priority.

2 Override the `Create` method (Delphi) or default constructor (C++) of the thread class by adding a new constructor to the declaration.
3 Code the constructor. The following is an example for a low-priority thread:

```
[Delphi]
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
```

```
[C++]
TMyThread::TMyThread( bool CreateSuspended ) : TThread( CreateSuspended ) {
  Priority = tpIdle;
}
```

4 Indicate whether the thread should be freed automatically when it finishes executing.

Warning: Boosting the thread priority of a CPU intensive operation may starve other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

To indicate when a thread is freed

1 Set the `FreeOnTerminate` property to true, unless the thread must be coordinated with other threads.

2 If the thread requires coordination with another thread, set `FreeOnTerminate` to false; then explicitly free the first thread from the second.

Iterating Through Strings in a List

This VCL application first creates a list of strings. Then it iterates through the strings, changing all string characters to uppercase. It consists of the following steps:

1. Create a VCL Form with Buttons and TListBox controls.
2. Write the code to create a string list and add strings to it.
3. Write the code to iterate through the string list to process string characters.
4. Run the application.

To create a VCL Form with TButton and TListBox controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons and a TListBox component on the form.
3. Select Button1 on the form.
4. In the **Object Inspector**, enter Add for the Name and Caption properties.
5. Select Button2 on the form.
6. In the **Object Inspector**, enter ToUpper for the Name and Caption properties.

To create a string list and add strings to it

1. Select the **Add** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.AddClick` (Delphi) or `TForm1::AddClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press ENTER.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created, and type the following variable declaration:

```
[Delphi]
var
  MyList: TStringList;
```

5. Insert the cursor within the code block, and type the following code:

```
[Delphi]
MyList := TStringList.Create;
    try
      with MyList do
      begin
        Add('Mice');
        Add('Goats');
        Add('Elephants');
        Add('Birds');
```

```
        ListBox1.Items.AddStrings(MyList);
      end;
    finally
      MyList.Free;
    end;
```

```
[C++]
TStringList *MyList = new TStringList();
try {
  MyList->Add("Mice");
  MyList->Add("Goats");
  MyList->Add("Elephants");
  MyList->Add("Birds");
  ListBox1->Items->AddStrings( MyList );
} __finally {
  MyList->Free();
}
```

To change all characters to uppercase

1. Select the **ToUpper** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.ToUpperClick` (Delphi) or `TForm1::ToUpperClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press return.

   This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

```
var
 Index: Integer;
```

5. Insert the cursor within the code block, and type the following code:

```
[Delphi]
for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
```

```
[C++]
for( int i = 0; i < ListBox1->Items->Count; i++ ) {
  ListBox1->Items[i] = UpperCase( ListBox1->Items[i] );
}
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application.

   The form displays with the controls.
2. Click the **Add** button.

The strings 'Mice', 'Goats', 'Elephants', and 'Birds' display in the order listed.

3 Click the **ToUpper** button.

The string characters display in uppercase.

Placing A Bitmap Image in a Control in a VCL Forms Application

This procedure adds a bitmap image to a combo box in a VCL forms application.

1. Create a VCL form.
2. Place components on the form.
3. Set component properties in the Object Inspector.
4. Write event handlers for the component's drawing action.
5. Build and run the application.

To create a VCL form with a TComboBox component

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Win32** page of the **Tool Palette**, place an TImageList component on the form.
3. From the **Standard** page of the **Tool Palette**, place a TComboBox on the form.

To set the component properties

1. Select ComboBox1 in the form.
2. In the **Object Inspector**, set the Style property drop-down to csOwnerDrawFixed.
3. In the **Object Inspector**, click the ellipsis next to the Items property.

   The **String List Editor** displays.
4. Enter a string you would like to associate with the bitmap image, for example, MyImage; then click **OK**.
5. Double-click ImageList1 in the form.

   The ImageList editor displays.
6. Click the **Add** button to display the **Add Images** dialog.
7. Locate a bitmap image to display in the Combo box.

   To locate an image, you can search for *.bmp images on your local drive. Select a very small image such as an icon. Copy it to your project directory, and click **Open**.

   The image displays in the ImageList editor.
8. Click **OK** to close the editor.

To add the event handler code

1. In the VCL form view, select ComboBox1.
2. In the Object Inspector, click the Events page, and double-click the OnDrawItem event.

   The **Code Editor** displays with cursor in the code block of the `ComboBox1DrawItem` (Delphi) or `ComboBox1::DrawItem` (C++) event handler.
3. Enter the following code for the event handler:

```
[Delphi]
Combobox1.Canvas.FillRect(Rect);
ImageList1.Draw(ComboBox1.Canvas, Rect.Left, Rect.Top, Index);
Combobox1.Canvas.TextOut(Rect.Left+ImageList1.Width+2,
  Rect.Top, ComboBox1.Items[Index]);
```

```
[C++]
ComboBox1->Canvas->FillRect( Rect );
ImageList1->Draw( ComboBox1->Canvas, Rect.Left, Rect.Top, Index );
ComboBox1->Canvas->TextOut( Rect.Left + ImageList1->Width + 2,
                            Rect.Top,
                            ComboBox1->Items[Index] );
```

To run the program

1. Choose **Run** ▸ **Run**.

   The applications executes, displaying a form with a combo box.

2. Click the combo box drop-down.

   The bitmap image and the text string display as a list item.

Reading a String and Writing It To a File

Creating this VCL application consists of the following steps:

1. Create a VCL Form with a button control.
2. Write the code to read the string and write it to a file.
3. Run the application.

To create a VCL Form

1. Create a directory in which to store your project files.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
3. From the **Standard** page of the Tool palette, place a TButton component on the form.

To read and write a string

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi. place the cursor before the `begin` reserved word; then press return.

   This creates a new line above the code block.
4. Type the following variable declarations:

```
[Delphi]
var fs: TFileStream;
const s: string = 'Hello';
```

```
[C++]
TFileStream *fs;
const AnsiString str;
```

5. Insert the cursor within the code block, and type the following code:

```
[Delphi]
fs := TFileStream.Create('temp.txt', fmCreate);
fs.Write(PChar(s)^, Length(s));
```

```
[C++]
fs = new TFileStream( "temp.txt", fmCreate );
fs->Write( (void*)str.c_str(), s.Length() );
```

To run the "Hello world" application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application.
   The form displays with a button called **Button1**.
2. Click **Button1**.
3. Use a text editor to open the newly created file temp.txt, which is located in your project directory.
   The string 'Hello' displays in the file.

Renaming Files

Creating this VCL application consists of the following steps:

1. Create a project directory containing a file to rename.
2. Create a VCL Form with button and label controls.
3. Write the code to rename the file.
4. Run the application.

To set up your project directory and a text file to copy

1. Create a directory in which to store your project files.
2. Either create or copy a text file to your project directory; then save it as MyFile.txt.

To create a VCL Form with a button and label

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton component on the form.
3. From the **Standard** page of the Tool palette, place a TLabel component on the form.

To write the rename file procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab.

   The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. At the cursor, type the following code:

```
if not RenameFile('MyFile.txt', 'YourFile.txt') then
Label1.Caption := 'Error renaming file!';
if( !RenameFile( "..\\MyFile.txt", "..\\YourFile.txt" )
  Label1->Caption = "Error renaming file";
// the file parameters assume the target output directory is in your project directory
```

Note: You cannot rename (move) a file across drives using RenameFile. You would need to first copy the file and then delete the old one. In the runtime library, RenameFile is a wrapper around the Windows API MoveFile function, so MoveFile will not work across drives either.

To run the application

1. Save your project file; then choose **Run** ▸ **Run** to build and run the application.

   The form displays.
2. Click the button; If no message displays in the Label, check the file name in your project directory.

MyFile.txt should is renamed as YourFile.txt.

3 If the caption label displays the error message, recheck your event handler code.

Using ActionManager to Create Actions in a VCL Forms Application

Using Developer Studio 2006, the following procedure illustrates how to create actions using ActionManager. It sets up a simple user interface with a text area, as would be appropriate for a text editing application, and describes how to create a file menu item with a file open action.

Building the VCL application with ActionManager actions consists of the following major steps:

1. Create a main window.
2. Add a File open action to the ActionManager.
3. Create the main menu.
4. Add the action to the menu.
5. Build and run the application.

To create a main window and add a File open action

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon.

   The **VCL Forms Designer** is displayed.
2. From the **Additional** page of the **Tool Palette**, add an TActionManager component to the form.
3. Double-click the TActionManager to display the Action Manager editor.

   Tip: To display captions for nonvisual components such as ActionManager, choose **Tools** ▸ **Environment Options**. On the **Designer** tab, check **Show component captions**, and click **OK**.
4. If necessary, click the **Actions** tab.
5. Select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog.
6. Scroll to the File category, and click the TFileOpen action.
7. Click **OK** to close the dialog.
8. In the Action Manager editor, select the File category.

   Open... displays in the **Actions:** list box.
9. Click **Close** to close the editor.

To create the main menu

1. From the **Additional** page of the **Tool Palette**, place an TActionMainMenuBar component on the form.
2. Open the Action Manager editor, and select the **File** category from the **Categories:** list box.
3. Drag File to the blank menu bar.

   File displays on the menu bar.

To build and run the application

1. Choose **Run** ▸ **Run**.

   The application executes, displaying a form with the main menu bar and the File menu.
2. Choose **File** ▸ **Open**.

The Open file dialog displays.

Using the Main VCL Thread

Using the main VCL thread consists of the following basic steps:

1. Create a separate routine to handle Windows messages received by components in your application.
2. Call CheckSynchronize periodically.
3. Declare thread-local variables, as necessary, for exclusive use by your thread.

To create a separate routine

1. Write a main thread routine that handles accessing object properties and executing object methods for all objects in your application.
2. Call the routine using the `TThread.Synchronize` (Delphi) or `TThread::Synchronize` method.

   The following code is an example of how to call a method using `Synchronize`

```
[Delphi]
procedure TMyThread.PushTheButton
begin
  Button1.Click;
end;
procedure TMyThread.Execute;
begin
  ...
  Synchronize(PushThebutton);
  ...
end;
```

```
[C++]
void TMyThread::PushTheButton() { Form1->Button1->Click(); }
void __fastcall TMyThread::Execute() {
  ...
  Synchronize( (TThreadMethod)&PushTheButton );
  ...
}
```

   `Synchronize` waits for the main thread to enter the message loop and then executes the passed method.

   Note: Because `Synchronize` uses a message loop, it does not work in console applications. For console applications, use other mechanisms, such as critical sections, to protect access to VCL objects.

To call CheckSynchronize

1. Call `CheckSynchronize` periodically within the main thread to enable background threads to synchronize execution with the main thread.
2. To ensure the safety of making background thread calls, call `CheckSynchronize` when the application is idle, for example, from an `OnIdle` event handler.

To use a thread-local variable

1. Identify variables that you want to make global to all the routines running in your thread but not shared by other instances of the same thread class.
2. For Delphi, declare these variables in a threadvar section, for example,

```
[Delphi]
threadvar
  x: integer;
```

   For C++, declare these variables with the __thread modifier:

```
[C++]
int __thread x;
```

Note: Use the threadvar section for global variables only. Do not use it for Pointer and Function variables or types that use copy-on-write semantics, such as long strings.

Note: For C++, if you initialize a __thread variable, you must initialize it to a constant expression. For example, `int __thread foo = 3;` is a legal statement, but `int __thread foo = get_value();` is not permitted because the initialization occurs at runtime.

Waiting for Threads

The following are procedures that can be used to wait for threads.

- Wait for a thread to finish executing.
- Wait for a task to complete.
- Check if another thread is waiting for your thread to terminate.

To wait for a thread to finish executing

1. Use the `WaitFor` method of the other thread.
2. Code your logic. For example, the following code waits for another thread to fill a thread list object before accessing the objects in the list:

```
[Delphi]
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
                for I := 0 to Count - 1 do
      ProcessItem(Items[I];
  end;
  ThreadList1.UnlockList;
end;
```

```
[C++]
if( ListFillingThread->WaitFor() ) {
  TList* list = ThreadList1->LockList();
  for( int i = 0; i < list->Count; i++ ) {
    DoSomething( list->Items[i] );
  }
  ThreadList1->UnlockList();
}
```

To wait for a task to complete

1. Create a TEvent object of global scope.
2. When a thread completes an operation other threads are waiting for, have the thread call `TEvent.SetEvent`.
3. To turn off the signal, call `TEvent.ResetEvent`.

The following example is an OnTerminate event handler that uses a global counter in a critical section to keep track of the number of terminating threads. When `Counter` reaches 0, the handler calls the `SetEvent` method to signal that all processes have terminated:

```
[Delphi]
procedure TDataModule.TaskTerminateThread(Sender: TObject);
begin
  ...
  CounterGuard.Acquire; {obtain a lock on the counter}
  Dec(Counter); {decrement the global counter variable}
```

```
  if Counter = 0 then
    Event1.SetEvent; {signal if this is the last thread}
  Counter.Release; {release the lock on the counter}
  ...
end;
```

```
[C++]
void __fastcall TDataModule::TaskThreadTerminate( TObject *Sender ) {
  ...
  CounterGuard->Acquire(); // lock the counter
  if( --Counter == 0 )      // decrement counter
    Event1->SetEvent();    // signal if this is the last thread
  CounterGuard->Release(); // release lock
}
```

The main thread initializes `Counter`, launches the task threads, and waits for the signal that they are all done by calling the `TEvent::WaitFor` method. `WaitFor` waits a specified time period for the signal to be set and returns one of the values in the table below.

The following code example shows how the main thread launches the task threads and resumes when they have completed.

```
[Delphi]
Event1.ResetEvent; {clear the event before launching the threads}
for i := 1 to Counter do
  TaskThread.Create(False); {create and launch the task threads}
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{continue with main thread}
```

```
[C++]
Event1->ResetEvent();  // clear the event before launching threads
for( int i = 0; i < Counter; i++ ) {
  new TaskThread( false );
if( Event1->WaitFor(20000) != wrSignaled )
  throw Exception;
// now continue with the main thread
```

Note: If you do not want to stop waiting for an event handler after a specified time period, pass the `WaitFor` method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

To check if another thread is waiting on your thread to terminate

1. In your `Execute` procedure, implement the `Terminate` method by checking and responding to the `Terminated` property.
2. This is one way to code the logic:

```
[Delphi]
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

```
[C++]
void __fastcall TMyThread::Execute() {
  while( !Terminated )
    DoSomething();
}
```

WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

Writing Cleanup Code

To clean up after your thread finishes executing

1. Centralize the cleanup code by placing it in the OnTerminate event handler.
   This ensures that the code gets executed.
2. Do not use any thread-local variables, because OnTerminate is not run as part of your thread.
3. You can safely access any objects from the OnTerminate handler.

Writing the Thread Function

The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program, because you must make sure that you do not overwrite memory that is used by other processes in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

To implement Execute, coordinate thread execution by

1. Synchronizing with a main VCL thread.
2. Avoiding simultaneous access to the same memory.
3. Waiting for threads.
4. Handling exceptions.

WebSnap Procedures

Building a WebSnap "Hello World" Application

Though simple, the WebSnap "Hello world" application demonstrates the essential steps for creating an WebSnap application.

Building the WebSnap "Hello world" application consists of five major steps:

1. Create a WebSnap project.
2. Accept the default included components.
3. Set the page title in the page options.
4. Modify the HTML template.
5. Run the application.

To create a WebSnap project

1. Choose **File** ▸ **New** ▸ **Other**.

   The **New Items** dialog appears.
2. In the **New Items** dialog, select **WebSnap Application** from the **Delphi Projects** ▸ **WebSnap** or **C++Builder Projects** ▸ **WebSnap** folder.
3. Click **OK**.

   The **New WebSnap Application** dialog appears.
4. Select the **Web App Debugger executable** radio button.
5. In the **Class Name** field, enter HelloWorld.
6. Select your application model components.
7. In the **Page Name** field, enter HelloWorld.
8. Select your caching type from the **Caching** drop-down.

To change included components (optional)

1. In the **New WebSnap Application** dialog, click **Components**.

   The **WebApp Components** dialog appears.
2. Select the components you want to include.

   Tip: In most cases, the default settings will suffice.
3. Click **OK**.

To set the page title in the page options

1. In the **New WebSnap Application** dialog, click **Page Options**.

   The **WebApp Components** dialog appears.
2. In the **Title** field, enter Hello World!.

To modify the HTML template

1. Click on the HTML tab in the IDE.
2. Below the line `<h2><%= Page.Title %></h2>`, insert a line saying `This is my first WebSnap application.`
3. Save the application.

To run the "Hello world" application

1. Choose **Run** ▸ **Run**.

   An application window opens, and the COM server registers your WebSnap application with the Web Application Debugger.
2. Close the application window.
3. Choose **Tools** ▸ **Web App Debugger** .

   The Web Application Debugger launches.
4. In the Web App Debugger, click the **Start** button.
5. Click on the **Default URL** to launch the browser.
6. In the browser, select your Hello World application from the list of applications and click **Go**.

   Your application appears in the browser with the text `Hello World! This is my first WebSnap application.`
7. Close the Web browser to return to the IDE.

Building a WebSnap Application

The following procedure describes the generic steps required to build a simple WebSnap project. For more advanced topics, refer to related information following the procedure.

Building a WebSnap application consists of five major steps:

1. Create an WebSnap project.
2. Change included components (optional).
3. Set page options (optional)
4. Create additional WebSnap pages.
5. Run the application.

To create a WebSnap project

1. Choose **File** ▸ **New** ▸ **Other**.
   The **New Items** dialog appears.
2. In the **New Items** dialog, select **WebSnap Application** from the **Delphi Projects** ▸ **WebSnap** folder.
3. Click **OK**.
   The **New WebSnap Application** dialog appears.
4. Select the type of application you are creating.
5. Select your application model components.
6. In the **Page Name** field, enter the name of your page.
7. Select your caching type from the **Caching** drop-down.

To change included components (optional)

1. In the **New WebSnap Application** dialog, click **Components**.
   The **WebApp Components** dialog appears.
2. Select the components you want to include.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.

To set page options (optional)

1. In the **New WebSnap Application** dialog, click **Page Options**.
   The **WebApp Components** dialog appears.
2. Set the page options.

To create additional WebSnap pages

1. In the **New Items** dialog, select **WebSnap Page Module** from the **Delphi Projects** ▸ **WebSnap** folder.
2. Configure the page module options and click **OK**.

3 Add and configure components.

Debugging a WebSnap Application using the Web Application Debugger

To debug a WebSnap Application using the Web Application Debugger

1. Register the server information application for the Web Application Debugger.
2. Register your WebSnap application with the Web Application Debugger the first time you run it.
3. Launch the Web Application Debugger.
4. Select and launch your web application.
5. Debug your web application using breakpoints and the Web Application Debugger log.

To register the server information application for the Web Application Debugger

1. Navigate to the bin directory of your Developer Studio 2006 installation.
2. Run `serverinfo.exe`.
3. Close the blank application window that opens.

This step only needs to be performed the first time you use the Web Application Debugger.

To register your web application with the Debugger

1. Choose **Run** ▸ **Run**.
   This displays the console window of the COM server that is your Web server application.
2. Close the blank application window that opens.

Your COM server is now registered so that the Web App debugger can access it.

To launch the Web Application Debugger

1. Choose **Tools** ▸ **Web App Debugger** .
   The Web Application Debugger launches.
2. In the Web App Debugger, click the **Start** button.
3. Click on the **Default URL** to launch the browser.

To select and launch your web application

1. In the browser, select your application from the list of applications.
2. Click **Go**.
   Your application appears in the browser.

Developer's Guide

Win32

Win32 Developer's Guide

Programming with Delphi

Delphi programming fundamentals

Designing Applications

You can design any kind of 32-bit applicationfrom general—purpose utilities to sophisticated data access programs or distributed applications.

As you visually design the user interface for your application, the Form Designer generates the underlying Delphi code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment.

You can create your own components using the Delphi language. Most of the components provided are written in Delphi. You can add components that you write to the **Tool palette** and customize the palette for your use by including new tabs if needed.

You can also design applications that run on both Linux and Windows by using CLX components. CLX contains a set of classes that, if used instead of those in the VCL, allows your program to port between Windows and Linux. If you are using Kylix while developing cross-platform applications, Kylix also includes a *Developer's Guide* that is tailored for the Linux environment. You can refer to the manual both in the Kylix online Help or the printed manual provided with the Kylix product.

Creating applications introduces support for different types of applications.

Creating Projects

All application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the **Project Manager**. The **Project Manager** lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools.

At the top of the project hierarchy is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the **Project Manager**. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the **Project Manager**, the project file is updated. You specify project options using a Project Options dialog which has tabs for

various aspects of your project such as forms, application, and compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of an application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

Editing Code

The Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the **Object Inspector**. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changes and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor, and continue adjusting the form from there.

The code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the VCL objects, CLX objects, RTL sources, and project files—can be viewed and edited in the Code editor.

Compiling Applications

When you have finished designing your application interface on the form and writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to build when you"ve changed global compiler directives to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles the changed files and any files that depend on them.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose **Project** ▸ **Compile All Projects**, or **Project** ▸ **Build All Projects** with the project group selected in the **Project Manager**.

Note: To compile a CLX application on Linux, you need Kylix.

Debugging Applications

With the integrated debugger, you can find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the variable values, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions are classes, like other classes in Delphi, except, by convention, they begin with an initial E rather than a T.

Deploying Applications

Delphi includes add-on tools to help with application deployment. For example, InstallShield Express (not available in all editions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. TeamSource software (not available in all editions) is also available for tracking application updates.

To deploy a CLX application on Linux, you need Kylix.

Note: Not all editions have deployment capabilities.

Refer to Deploying Applications for specific information on deployment.

Understanding the component library

Understanding the Component Library

The component library is made up of objects separated into several sublibraries, each of which serves a different purpose. These sublibraries are listed in the following table:

Component sublibraries

Part	Description
VCL/RTL	Low-level classes and routines available for all VCL applications. VCL/RTL includes the runtime library (RTL) up to and including the Classes unit.
DataCLX	Client data-access components. The components in DataCLX are a subset of the total available set of components for working with databases. These components are used in cross-platform applications that access databases. They can access data from a file on disk or from a database server using dbExpress.
NetCLX	Components for building Web Server applications. These include support for applications that use Apache or CGI Web Servers.
VisualCLX	Cross-platform GUI components and graphics classes. VisualCLX classes make use of an underlying cross-platform widget library (Qt).
WinCLX	Classes that are available only on the Windows platform. These include controls that are wrappers for native Windows controls, database access components that use mechanisms (such as the Borland Database Engine or ADO) that are not available on Linux, and components that support Windows-only technologies (such as COM, NT Services, or control panel applets).

The VCL and CLX contain many of the same sublibraries. They both include BaseCLX, DataCLX, NetCLX. The VCL also includes WinCLX while CLX includes VisualCLX instead. Use the VCL when you want to use native Windows controls, Windows-specific features, or extend an existing VCL application. Use CLX when you want to write a cross-platform application or use controls that are available in CLX applications, such as *TLCDNumber*.

All classes descend from *TObject*. *TObject* introduces methods that implement fundamental behavior like construction, destruction, and message handling.

Components are a subset of the component library that descend from the class *TComponent*. You can place components on a form or data module and manipulate them at design time. Using the **Object Inspector**, you can assign property values without writing code. Most components are either visual or nonvisual, depending on whether they are visible at runtime. Some components appear on the **Tool palette**.

Visual components, such as *TForm* and *TSpeedButton*, are called *controls* and descend from *TControl*. Controls are used in GUI applications, and appear to the user at runtime. *TControl* provides properties that specify the visual attributes of controls, such as their height and width.

Nonvisual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a *TDataSource* component on a form to connect a control and a dataset used by the

control. This connection is not visible to the user, so *TDataSource* is nonvisual. At design time, nonvisual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control.

Classes that are not components (that is, classes that descend from *TObject* but not *TComponent*) are also used for a variety of tasks. Typically, these classes are used for accessing system objects (such as a file or the clipboard) or for transient tasks (such as storing data in a list). You can't create instances of these classes at design time, although they are sometimes created by the components that you add in the Form Designer.

Detailed reference material on all VCL and CLX objects is accessible while you are programming. In the Code editor, place the cursor anywhere on the object and press F1 to display the Help topic. Objects, properties, methods, and events that are in the VCL are marked "VCL Reference" and those in CLX are marked "CLX Reference."

Properties, Methods, and Events

Both the VCL and CLX form hierarchies of classes that are tied to the IDE, where you can develop applications quickly. The classes in both component libraries are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The component library is written in the Delphi language, although the VCL is based on the Windows API and CLX is based on the Qt widget library.

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change some properties at design time and get immediate feedback as the components change in the IDE.
- You can access some properties in the **Object Inspector**, where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The calls to get and set the values of properties can be methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of a property. For example, changing the value of one property may require you to modify another. You can change the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing one property on one object can affect several objects. For example, setting the *Checked* property on a radio button affects all of the radio buttons in the group.

Methods

A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the public, protected, and private properties and fields of the class and are commonly referred to as member functions. Although most methods belong to an instance of a class, some methods belong instead to the class type. These are called class methods.

Events

An *event* is an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform. For example, the user may choose a menu item, click a button, or mark some text. You can write code to handle the events in which you are interested, rather than writing code that always executes in the same restricted order.

Regardless of how an event is triggered, VCL objects look to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

Types of Events

The kinds of events that can occur can be divided into two main categories:

- User events
- System events
- Internal events

User events

User events are actions that the user initiates. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDblClick* (the user double-clicked a mouse button).

System events

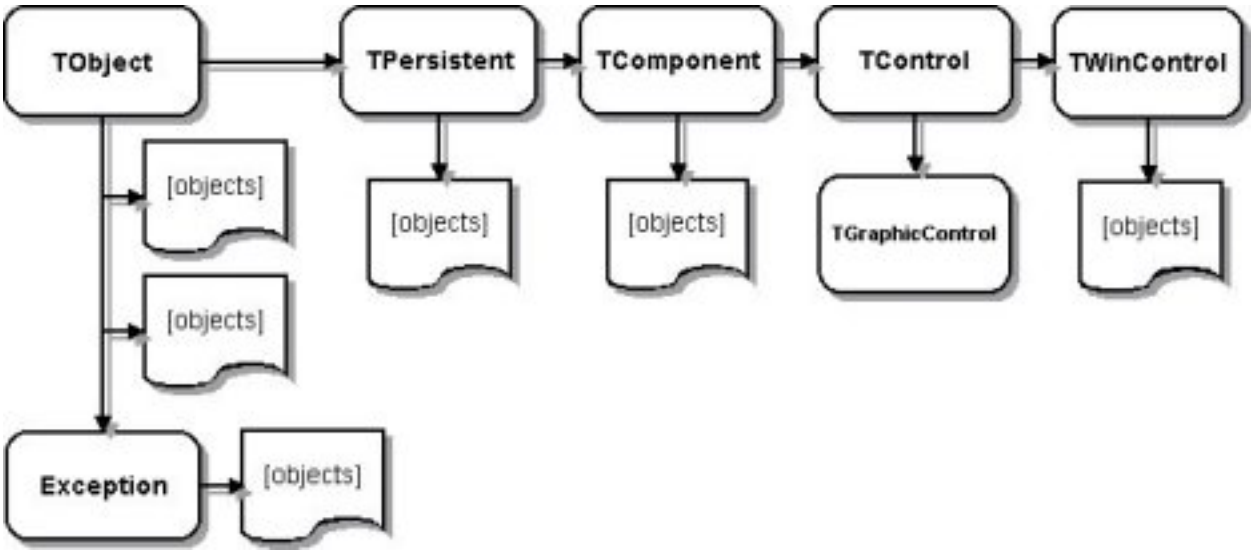
System events are events that the operating system fires for you. For example, the *OnTimer* event (which the Timer component issues whenever a predefined interval has elapsed), the *OnPaint* event (a component or window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

Internal events

Internal events are events that are generated by the objects in your application. An example of an internal event is the *OnPost* event that a dataset generates when your application tells it to post the current record.

Objects, Components, and Controls

The following diagram is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.



Every object (class) inherits from *TObject*. Objects that can appear in the Form Designer inherit from *TPersistent* or *TComponent*. Controls, which appear to the user at runtime, inherit from *TControl*. There are two types of controls, graphic controls, which inherit from *TGraphicControl*, and windowed controls, which inherit from *TWinControl* or *TWidgetControl*. A control like *TCheckBox* inherits all the functionality of *TObject*, *TPersistent*, *TComponent*, *TControl*, and *TWinControl* or *TWidgetControl*, and adds specialized capabilities of its own.

The figure shows several important base classes, which are described in the following table:

Class	Description
TObject	Signifies the base class and ultimate ancestor of everything in the VCL or CLX. *TObject* encapsulates the fundamental behavior common to all VCL/CLX objects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
Exception	Specifies the base class of all classes that relate to VCL exceptions. *Exception* provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
TPersistent	Specifies the base class for all objects that implement publishable properties. Classes under *TPersistent* deal with sending data to streams and allow for the assignment of classes.
TComponent	Specifies the base class for all components. Components can be added to the **Tool palette** and manipulated at design time. Components can own other components.
TControl	Represents the base class for all controls that are visible at runtime. *TControl* is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
TWinControl or TWidgetControl	Specifies the base class of all controls that can have keyboard focus. Controls under *TWinControl* are called windowed controls while those under *TWidgetControl* are called widgets.

For a complete overview of the VCL and CLX object hierarchies, refer to the VCL Object Hierarchy and CLX Object Hierarchy wall charts included with this product.

TObject Branch

The *TObject* branch includes all VCL and CLX classes that descend from *TObject* but not from *TPersistent*. Much of the powerful capability of the component library is established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all classes in the component library by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.
- Support for handling messages (VCL applications) .

TObject is the immediate ancestor of many simple classes. Classes in the *TObject* branch have one common, important characteristic: they are transitory. This means that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another group in the *TObject* branch is classes that encapsulate data structures, such as:

- *TBits*, a class that stores an "array" of Boolean values.
- *TList*, a linked list class.
- *TStack*, a class that maintains a last-in first-out array of pointers.
- *TQueue*, a class that maintains a first-in first-out array of pointers.

Another group in the *TObject* branch are wrappers for external objects like *TPrinter*, which encapsulates a printer interface, and *TIniFile*, which lets a program read from or write to an ini file.

TStream is a good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on (see Using streams for information on streams).

TPersistent Branch

The *TPersistent* branch includes all VCL and CLX classes that descend from *TPersistent* but not from *TComponent*. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Because of their persistence, objects from this branch can appear at design time. However, they can't exist independently. Rather, they implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. *TPersistent* introduces the *GetOwner* method, which lets the Form Designer determine the owner of the object.

Classes in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method lets each class indicate how to load and save properties.

Following are some of the classes in the *TPersistent* branch of the hierarchy:

- Graphics such as: *TBrush*, *TFont*, and *TPen*.
- Classes such as *TBitmap* and *TIcon*, which store and display visual images, and *TClipboard*, which contains text or graphics that have been cut or copied from an application.
- String lists, such as *TStringList*, which represent text or lists of strings that can be assigned at design time.
- Collections and collection items, which descend from *TCollection* or *TCollectionItem*. These classes maintain indexed collections of specially defined items that belong to a component. Examples include *THeaderSections* and *THeaderSection* or *TListColumns* and *TListColumn*.

TComponent Branch

The *TComponent* branch contains classes that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time but which do not appear to the user at runtime. They are persistent objects that can do the following:

- Appear on the **Tool palette** and be changed on the form.
- Own and manage other components.
- Load and save themselves.

Several methods introduced by *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming (the saving and loading of form files, which store information about the property values of objects on a form) is introduced in this branch. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* branch also introduces the concept of ownership that is propagated throughout the component library. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Components* property.

The constructor for every component takes a parameter that specifies the new component's owner. If the passed-in owner exists, the new component is added to that owner's Components list. Aside from using the Components list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by a form are destroyed and their memory freed when the form is destroyed. (Assuming, of course, that the components have properly designed destructors that clean them up correctly.)

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and reads values for that instance's properties.

Some of the classes in the *TComponent* branch include:

- *TActionList*, a class that maintains a list of actions, which provides an abstraction of the responses your program can make to user input.
- *TMainMenu*, a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TOpenDialog*, TSaveDialog, TFontDialog, TFindDialog, TColorDialog, and so on, classes that display and gather information from commonly used dialog boxes.
- *TScreen*, a class that keeps track of the forms and data modules that an application creates, the active form, the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the **Tool palette** but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

See Working with components for details on setting properties, calling methods, and working with events for components.

TControl Branch

The *TControl* branch consists of components that descend from *TControl* but not *TWinControl (TWidgetControl* in CLX applications*)*. Classes in this branch are controls: visual objects that the user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position,

the cursor associated with the control's window, methods to paint or move the control, and events to respond to mouse actions. Controls in this branch, however, can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

TControl introduces many visual properties that all controls inherit. These include the *Caption*, *Color*, *Font*, and *HelpContext* or *HelpKeyword*. While these properties inherited from *TControl*, they are only published—and hence appear in the **Object Inspector**—for controls to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays. *TControl* also introduces the *Parent* property, which specifies another control that visually contains the control.

Classes in the *TControl* branch often called graphic controls, because they all descend from *TGraphicControl*, which is an immediate descendant of *TControl*. Although these controls appear to the user at runtime, graphic controls do not have their own underlying window or widget. Instead, they use their parent's window or widget. It is because of this limitation that graphic controls cant receive keyboard input or act as a parent to other controls. However, because they do not have their own window or widget, graphic controls use fewer system resources. For details on many of the classes in the *TControl* branch, see graphics controls.

There are two versions of *TControl*, one for VCL (Windows-only) applications and one for CLX (cross-platform) applications. Most controls have two versions as well, a Windows-only version that descends from the Windows-only version of *TControl*, and a cross-platform version that descends from the cross-platform version of *TControl*. The Windows-only controls use native Windows APIs in their implementations, while the cross-platform versions sit on top of the Qt cross-platform widget library.

TWinControl/TWidgetControl Branch

Most controls fall into the TWinControl/ *TWidgetControl* branch. Unlike graphic controls, controls in this branch have their own associated window or widget. Because of this, they are sometimes called windowed controls or widget controls. Windowed controls all descend from *TWinControl*, which descends from the windows-only version of *TControl*. Widget controls all descend from *TWidgetControl*, which descends from the CLX version of *TControl*.

Controls in the *TWinControl*/*TWidgetControl* branch:

- Can receive focus while an application is running, which means they can receive keyboard input from the application user. In comparison, graphic controls can only display data and respond to the mouse.
- Can be the parent of one or more child controls.
- Have a handle, or unique identifier, that allows them to access the underlying window or widget.

The *TWinControl/TWidgetControl* branch includes both controls that are drawn automatically (such as *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that do not correspond directly to a single underlying Windows control or widget. Controls in this latter category, which includes classes like *TStringGrid* and *TDBNavigator*, must handle the details of painting themselves. Because of this, they descend from *TCustomControl*, which introduces a *Canvas* property on which they can paint themselves.

Using the object model

Using the Object Model

The Delphi language is a set of object-oriented extensions to standard Pascal. Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you define a class, you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

The following topics provide a brief introduction to object-oriented concepts for programmers who are just starting out with the Delphi language. For more details on object-oriented programming for programmers who want to write components that can be installed on the **Tool palette**, see Overview of Component Creation.

- What is an object?
- Inheriting data and code from an object
- Scope and qualifiers
- Using object variables
- Creating, instantiating, and destroying objects
- Defining new classes
- Using interfaces

What Is an Object?

A *class* is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements. An *object* is an instance of a class. That is, it is a value whose type is a class. The term object is often used more loosely in this documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

You can begin to understand objects if you understand Pascal records or *structures* in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of many Delphi objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

Examining a Delphi Object

When you create a new project, the IDE displays a new form for you to customize. In the Code editor, the automatically generated unit declares a new class type for the form and includes the code that creates the new form instance. The generated code for a new Windows application looks like this:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
TForm1 = class(TForm){ The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end;{ The type declaration of the form ends here }
var
  Form1: TForm1;
implementation{ Beginning of implementation part }
{$R *.dfm}
end.{ End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added any components (the fields of the new object) to the form and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

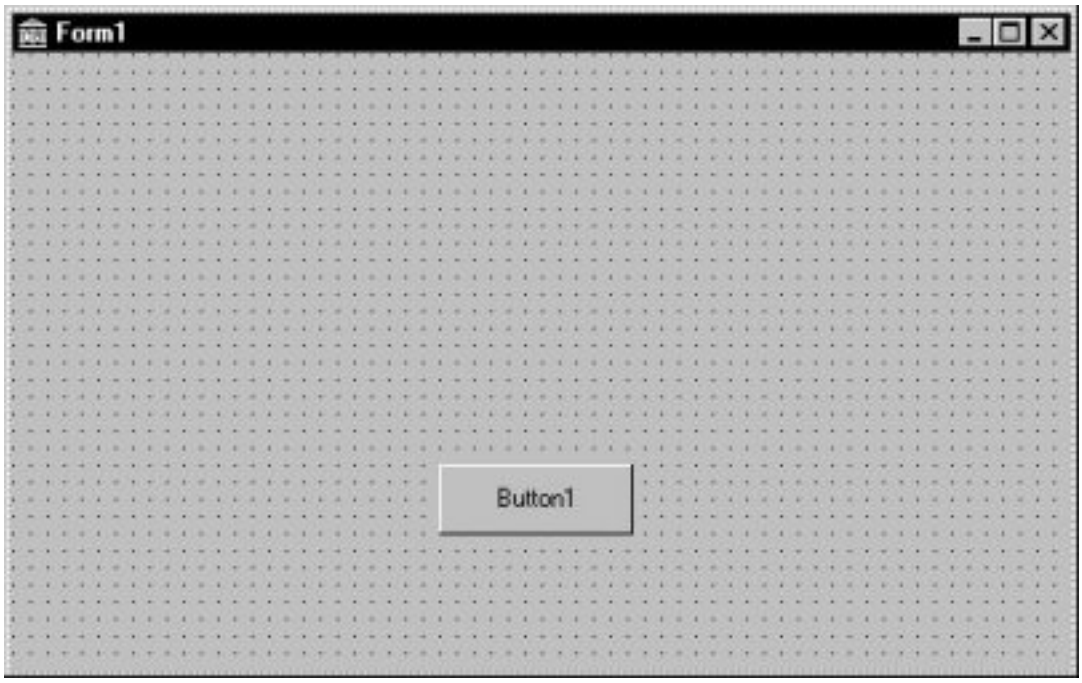
```
var
  Form1: TForm1;
```

Form1 represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete GUI application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
    Button1: TButton;{ New data field }
    procedure Button1Click(Sender: TObject);{ New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
begin
  Form1.Color := clGreen;
end;
end.
```

TForm1 has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write using the IDE are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared in the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the implementation part of the unit.

Changing the Name of a Component

You should always use the **Object Inspector** to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorWindow*. When you change the form's *Name* property in the **Object Inspector**, the new name is automatically reflected in the form's .dfm or .xfm file (which you usually don't edit manually) and in the source code that the IDE generates:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
  TColorWindow = class(TForm){ Changed from TForm1 to TColorWindow }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  ColorWindow: TColorWindow;{ Changed from Form1 to ColorWindow }
implementation
{$R *.dfm}
procedure TColorWindow.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn"t change! }
end;
end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorWindow.Button1Click(Sender: TObject);
begin
  ColorWindow.Color := clGreen;
end;
```

Inheriting Data and Code from an Object

The *TForm1* object in examining a Delphi object seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with TForm and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, the IDE automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived class inherits all the properties, events, and methods of the class from which it derives. The derived class is called a *descendant* and the class from which it derives is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. A Delphi class can have only one immediate ancestor, but it can have many direct descendants.

Scope and Qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared in a class are available to that class and, as is discussed later, often to its descendants. Although a method's implementation code appears outside of the class declaration, the method is still within the scope of the class because it is declared in the class declaration.

When you write code to implement a method that refers to properties, methods, or fields of the class where the method is declared, you don't need to preface those identifiers with the name of the class. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

The IDE creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the uses clause of *Form1*'s unit.

The scope of a class extends to its descendants. You can, however, redeclare a field, property, or method in a descendant class. Such redeclarations either hide or override the inherited member.

Private, Protected, Public, and Published Declarations

A class type declaration contains three or four possible sections that control the accessibility of its fields and methods:

```
Type
  TClassName = Class(TObject)
```

```
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

- The public section declares fields and methods with no access restrictions. Class instances and descendant classes can access these fields and methods. A public member is accessible from wherever the class it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.
- The protected section includes fields and methods with some access restrictions. A protected member is accessible within the unit where its class is declared and by any descendant class, regardless of the descendant class's unit.
- The private section declares fields and methods that have rigorous access restrictions. A private member is accessible only within the unit where it is declared. Private members are often used in a class to implement other (public or published) methods and properties.
- For classes that descend from *TPersistent*, a published section declares properties and events that are available at design time. A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the **Object Inspector** at design time.

When you declare a field, property, or method, the new member is added to one of these four sections, which gives it its *visibility*: private, protected, public, or published.

Using Object Variables

You can assign one object variable to another object variable if the variables are of the same type or are assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable to which you are assigning is an ancestor of the type of the variable being assigned. For example, here is a *TSimpleForm* type declaration and a variable declaration section declaring two variables, *AForm* and *Simple*:

```
type
  TSimpleForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  AForm: TForm;
  SimpleForm: TSimpleForm;
```

AForm is of type *TForm*, and *SimpleForm* is of type *TSimpleForm*. Because *TSimpleForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := SimpleForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
.
.
.
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word is. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

Creating, Instantiating, and Destroying Objects

Many of the objects you use in the Form Designer, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and data source components, have no visual representation at runtime.

You may want to create your own classes. For example, you could create a *TEmployee* class that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the interface or implementation part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically. However, if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a

destructor) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a nil reference before calling *Destroy*. For example,

```
Employee.Free;
```

destroys the *Employee* object and deallocates its memory.

Components and Ownership

Delphi components have a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

Defining New Classes

Although there are many classes in the object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
  TClassName = Class (TParentClass)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

To define a class:

1. In the IDE, start with a project open and choose **File** ▸ **New** ▸ **Unit** to create a new unit where you can define the new class.
2. Add the uses clause and type section to the interface section.
3. In the type section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
.
.
.
private
.
.
.
published {If descended from TPersistent or below}
.
.
.
```

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

4 Some editions of the IDE include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare. If you have code completion, invoke it to finish the class declaration: place the cursor within a method definition in the interface section and press Ctrl+Shift+C (or right-click and select Complete Class at Cursor). Any unfinished property declarations are completed, and for any methods that require an implementation, empty methods are added to the implementation section.

If you do not have class completion, you need to write the code yourself, completing property declarations and writing the methods.

Given the example above, if you have class completion, read and write specifiers are added to your declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

The following code is also added to the implementation section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
end;
procedure TMyButton.SetSize(const Value: Integer);
begin
FSize := Value;
end;
```

5 Fill in the methods. For example, to make it so the button beeps when you call the *DoSomething* method, add the Beep between begin and end.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;
procedure TMyButton.SetSize(const Value: Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    DoSomething;
  end;
end;
```

Note that the button also beeps when you call *SetSize* to change the size of the button.

Using Interfaces

Delphi is a single-inheritance language. That means that any class has only a single direct ancestor. However, there are times you want a new class to inherit properties and methods from more than one base class so that you can use it sometimes like one and sometimes like the other. Interfaces let you achieve something like this effect.

An interface is like a class that contains only abstract methods (methods with no implementation) and a clear definition of their functionality. Interface method definitions include the number and types of their parameters, their return type, and their expected behavior. By convention, interfaces are named according to their behavior and prefaced with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  procedure MyProcedure;
end;
```

A simple example of an interface declaration is:

```
type
IEdit = interface
  procedure Copy;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
end;
```

Interfaces can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy;
  procedure Cut;
```

```
  procedure Paste;
  function Undo: Boolean;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

Using Interfaces Across the Hierarchy

Using interfaces lets you separate the way a class is used from the way it is implemented. Two classes can implement the same interface without descending from the same base class. By obtaining an interface from either class, you can call the same methods without having to know the type of the class. This polymorphic use of the same methods on unrelated objects is possible because the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from a common ancestor (say, *TFigure*), which declares a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. In the previous example, *IPaint* could be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

IRotate makes sense for the rectangle but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;
```

```
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern that fills the circle without having to add rotation to the simple circle.

Note: For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface*, the base interface from which all interfaces descend. For more information on *IInterface*, see Implementing IInterface and Memory management of interface objects.

Using Interfaces with Procedures

Interfaces allow you to write generic procedures that can handle objects without requiring that the objects descend from a particular base class. Using the *IPaint* and *IRotate* interfaces defined previously, you can write the following procedures:

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;
procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate. This allows the generic procedures to be used more often than if they were written to only work against a *TFigure* class.

Implementing IInterface

Just as all objects descend, directly or indirectly, from *TObject*, all interfaces derive from the *IInterface* interface. IInterface provides for dynamic querying and lifetime management of the interface. This is established in the three *IInterface* methods:

- *QueryInterface* dynamically queries a given object to obtain interface references for the interfaces that the object supports.
- *_AddRef* is a reference counting method that increments the count each time a call to QueryInterface succeeds. While the reference count is nonzero the object must remain in memory.
- *_Release* is used with _AddRef to allow an object to track its own lifetime and determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory. Every class that implements interfaces must implement the three IInterface methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of lifetime management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

TInterfacedObject

When defining a class that supports one or more interfaces, it is convenient to use *TInterfacedObject* as a base class because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
procedure BeforeDestruction; override;
class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
  .
  .
  .
  end;
```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see Memory management of interface objects, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

Using the as Operator with Interfaces

Classes that implement interfaces can use the as operator for dynamic binding on the interface. In the following example,

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;
begin
  X := P as IPaint;
{ statements }
end;
```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X,* which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an *IPaint* reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was of a class type that did not implement *IInterface*.

When you use the as operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the as operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the as operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the Ctrl+Shift+G editor shortcut key.

Reusing Code and Delegation

One approach to reusing code with interfaces is to have one interfaced object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, the Delphi language has a keyword implements, that makes if easy to write code to delegate all or part of the implementation of an interface to a subobject.

Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object uses an inner object that implements interfaces which are exposed only by the outer object.

Using Implements for Delegation

Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword implements to specify that the methods of that interface are delegated to the object or interface reference which is the value of the property. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default, using the implements keyword delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods to override this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;
interface
type
IRGB8bit = interface
['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;
IColorRef = interface
['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;
{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
```

```
  end;
implementation
constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;
function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that make up the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. At a minimum, it must implement *IInterface*. The inner object, or objects, also implement one or more interfaces. However, only the outer object exposes the interfaces. That is, the outer object exposes both the interfaces it implements and the ones that its contained objects implement.

Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The aggregation model defines explicit rules for implementing *IInterface* using delegation. The inner object must implement two versions of the *IInterface* methods.

- It must implement *IInterface* on itself, controlling its own reference count. This implementation of *IInterface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IInterface*.
- It also implements a second *IInterface* for all the interfaces it implements that the outer object exposes. This second *IInterface* delegates calls to *QueryInterface*, *_AddRef*, and *_Release* to the outer object. The outer *IInterface* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *IInterface* in *TComObject*. *TComObject* is a COM class that supports aggregation. If you are writing COM applications, you can also use *TComObject* directly as a base class.

Memory Management of Interface Objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way to implement this lifetime management. *_AddRef* and *_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

Using Reference Counting

The Delphi compiler provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from *TInterfacedObject*. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);
function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
  // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
  // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

Note: In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *_AddRef*) on the parameter, whereas either of the following declarations do not:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

Not Using Reference Counting

If your object is a component or a control that is owned by another component, then it is part of a different memory management system that is based in *TComponent*. Although some classes mix the object lifetime management approaches of *TComponent* and interface reference counting, this is very tricky to implement correctly.

To create a component that supports interfaces but bypasses the interface reference counting mechanism, you must implement the *_AddRef* and *_Release* methods in code such as the following:

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;
function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you implement *QueryInterface*, you can still use the as operator for interfaces, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the "controlling Unknown." It is at the level of the outer object that the decision is made to circumvent the *_AddRef* and *_Release* methods, and to handle memory management via another approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a as its containing outer object one that does not follow the interface lifetime model.

Note: The "controlling Unknown" is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *IInterface*, but is used instead in COM-based applications (Windows only). For more information distinguishing the various implementations of the *IUnknown* or *IInterface* interface by the inner and outer objects, see Aggregation and the Microsoft online Help topics on the "controlling Unknown."

Using Interfaces in Distributed Applications

In VCL applications, interfaces are a fundamental element in the COM, SOAP, and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods.

When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of *QueryInterface*, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see Overview of COM technologies.

When distributing an application using SOAP, interfaces are required to carry their own runtime type information (RTTI). The compiler only adds RTTI to an interface when it is compiled using the {$M+} switch. Such interfaces are called *invokable interfaces*. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, client applications can only call the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be called by clients.

The easiest way to define invokable interfaces is to define your interface so that it descends from *IInvokable*. *IInvokable* is the same as *IInterface*, except that it is compiled using the {$M+} switch. For more information about Web Service applications that are distributed using SOAP, and about invokable interfaces, see Using Web Services.

Using the VCL/RTL

Using the VCL/RTL: Overview

There are a number of units in the component library that provide the underlying support for most of the component libraries. These units include the global routines that make up the runtime library, a number of utility classes such as those that represent streams and lists, and the classes *TObject*, *TPersistent*, and *TComponent*. Collectively, these units are called the VCL/RTL. The VCL/RTL does not include any of the components that appear on the **Tool Palette**. Rather, the classes and routines in the VCL/RTL are used by the components that do appear on the **Tool Palette** and are available for you to use in application code or when you are writing your own classes.

The following topics discuss many of the classes and routines that make up the VCL/RTL and illustrate how to use them.

- Using streams
- Working with files
- Working with .ini files
- Working with lists
- Working with string lists
- Working with strings
- Creating drawing spaces
- Printing
- Converting measurements
- Defining custom variants

Note: This list of tasks is not exhaustive. The runtime library in the VCL/RTL contains many routines to perform tasks that are not mentioned here. These include a host of mathematical functions (defined in the Math unit), routines for working with date/time values (defined in the SysUtils and DateUtils units), and routines for working with Variant values (defined in the Variants unit).

Using Streams

Streams are classes that let you read and write data. They provide a common interface for reading and writing to different media such as memory, strings, sockets, and BLOB fields in databases. There are several stream classes, which all descend from *TStream*. Each stream class is specific to one media type. For example, *TMemoryStream* reads from or writes to a memory image; *TFileStream* reads from or writes to a file.

The following topics describe the methods common to all stream classes:

- Using streams to read or write data
- Copying data from one stream to another
- Specifying the stream position and size

Using Streams to Read or Write Data

Stream classes all share several methods for reading and writing data. These methods are distinguished by whether they:

- Return the number of bytes read or written.
- Require the number of bytes to be known.
- Raise an exception on error.

Stream methods for reading and writing

The *Read* method reads a specified number of bytes from the stream, starting at its current *Position*, into a buffer. *Read* then advances the current position by the number of bytes actually transferred. The prototype for *Read* is:

```
[Delphi]
function Read(var Buffer; Count: Longint): Longint;
```

```
[C++]
virtual int __fastcall Read(void *Buffer, int Count);
```

Read is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the stream did not contain *Count* bytes of data past the current position.

The *Write* method writes *Count* bytes from a buffer to the stream, starting at the current *Position*. The prototype for *Write* is:

```
[Delphi]
function Write(const Buffer; Count: Longint): Longint;
```

```
[C++]
virtual int __fastcall Write(const void *Buffer, int Count);
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered or the stream can't accept any more bytes.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception (*EReadError* and *EWriteError*) if the byte count can not be matched exactly. This is in contrast to the *Read* and *Write* methods, which can return a byte count that differs from the requested value. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
[Delphi]
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

```
[C++]
virtual int __fastcall ReadBuffer(void *Buffer, int Count);
virtual int __fastcall WriteBuffer(const void *Buffer, int Count);
```

These methods call the *Read* and *Write* methods to perform the actual reading and writing.

Reading and writing components

TStream defines specialized methods, ReadComponent and WriteComponent, for reading and writing components. You can use them in your applications as a way to save components and their properties when you create or alter them at runtime.

ReadComponent and *WriteComponent* are the methods that the IDE uses to read components from or write them to form files. When streaming components to or from a form file, stream classes work with the *TFiler* classes, *TReader* and *TWriter*, to read objects from the form file or write them out to disk. For more information about using the component streaming system, see TStream, TFiler, TReader, TWriter, and TComponent classes.

Reading and writing strings

If you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in the pointer element. You need to first cast the string to a *Pointer* or *PChar*, and then dereference it. For example:

```
[Delphi]
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// this will give you garbage
  fs.Write(PChar(s)^, Length(s));// this is the correct way
end;
```

Copying Data from One Stream to Another

When copying data from one stream to another, you do not need to explicitly read and then write the data. Instead, you can use the *CopyFrom* method, as illustrated in the following example.

In the following example, one file is copied to another one using streams. The application includes two edit controls (*EdFrom* and *EdTo*) and a *Copy File* button.

```
[Delphi]
procedure TForm1.CopyFileClick(Sender: TObject);
var
Source, Destination:TStream;
begin
Source := TFileStream.Create(edFrom.Text, fmOpenRead or fmShareDenyWrite);
try
Destination := TFileStream.Create(edTo.Text, fmOpenCreate or fmShareDenyRead);
try
Destination.CopyFrom(Source,Source.Size);
    finally
Destination.Free;
    end;
  finally
    Source.Free
end;
```

```
[C++]
void __fastcall TForm1::CopyFileClick(TObject *Sender)
{
TStream* Source= new TFileStream(edFrom->Text, fmOpenRead | fmShareDenyWrite);
try
  {
    TStream* Destination = new TFileStream(edTo->Text, fmCreate | fmShareDenyRead);
try
    {
      Destination -> CopyFrom(Source, Source->Size);
    }
    __finally
    {
      delete Destination;
    }
  }
  __finally
  {
    delete Source;
  }
}
```

Specifying the Stream Position and Size

In addition to methods for reading and writing, streams permit applications to seek to an arbitrary position in the stream or change the size of the stream. Once you seek to a specified position, the next read or write operation starts reading from or writing to the stream at that position.

Seeking to a specific position

The *Seek* method is the most general mechanism for moving to a particular position in the stream. There are two overloads for the *Seek* method:

```
[Delphi]
function Seek(Offset: Longint; Origin: Word): Longint;
function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
```

```
[C++]
virtual int __fastcall Seek(int Offset, Word Origin);
virtual __int64 __fastcall Seek(const __int64 Offset, TSeekOrigin Origin);
```

Both overloads work the same way. The difference is that one version uses a 32-bit integer to represent positions and offsets, while the other uses a 64-bit integer.

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Values for the Origin parameter

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file.

Seek resets the current stream position, moving it by the indicated offset. *Seek* returns the new current position in the stream.

Using Position and Size properties

All streams have properties that hold the current position and size of the stream. These are used by the *Seek* method, as well as all the methods that read from or write to the stream.

The *Position* property indicates the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
[Delphi]
property Position: Int64;
```

```
[C++]
__property __int64 Position = {read=GetPosition, write=SetPosition, nodefault};
```

The *Size* property indicates the size of the stream in bytes. It can be used to determine the number of bytes available for reading, or to truncate the data in the stream. The declaration for *Size* is:

```
[Delphi]
property Size: Int64;
```

```
[C++]
__property __int64 Size = {read=GetSize, write=SetSize64, nodefault};
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the data in the stream. For example, on a file stream, setting *Size* inserts an end of file marker to truncate the file. If the *Size* of the stream cannot be changed, an exception is raised. For example, trying to change the *Size* of a read-only file stream raises an exception.

Working with Files

The VCL/RTL supports several ways of working with files. In addition to using file streams, there are several runtime library routines for performing file I/O. Both file streams and the global routines for reading from and writing to files are described in Approaches to file I/O.

In addition to input/output operations, you may want to manipulate files on disk. Support for operations on the files themselves rather than their contents is described in Manipulating files.

Note: When writing cross-platform applications, remember that although the Delphi language is not case sensitive, the Linux operating system is. When using objects and routines that work with files, be attentive to the case of file names.

Approaches to File I/O

There are several approaches you can take when reading from and writing to files:

- The recommended approach for working with files is to use file streams. File streams are instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. Because file streams make the file handle available, this approach can be combined with the next one. The Using file streams discusses *TFileStream* in detail.
- You can work with files using a handle-based approach. File handles are provided by the operating system when you create or open a file to work with its contents. The SysUtils unit defines a number of file-handling routines that work with files using file handles. On Windows, these are typically wrappers around Windows API functions. Because the VCL/RTL functions can use the Delphi language syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Furthermore, there are corresponding versions on Linux, so you can use these routines in cross-platform applications. To use a handle-based approach, you first open a file using the *FileOpen* function or create a new file using the *FileCreate* function. Once you have the handle, use handle-based routines to work with its contents (write a line, read text, and so on).
- The System unit defines a number of file I/O routines that work with file variables, usually of the format "F: Text: " or "F: File:" File variables can have one of three types: typed, text, and untyped. A number of file-handling routines, such as *AssignPrn* and *writeln*, use them. The use of file variables is deprecated, and these file types are supported only for backward compatibility. They are incompatible with Windows file handles.

Using File Streams

The *TFileStream* class enables applications to read from and write to a file on disk. Because *TFileStream* is a stream object, it shares the common stream methods. You can use these methods to read from or write to the file, copy data to or from other stream classes, and read or write components values. See Using streams for details on the capabilities that files streams inherit by being stream classes.

In addition, file streams give you access to the file handle, so that you can use them with global file handling routines that require the file handle.

Creating and opening files using file streams

To create or open a file and get access to its handle, you simply instantiate a *TFileStream*. This opens or creates a specified file and provides methods to read from or write to it. If the file cannot be opened, the *TFileStream* constructor raises an exception.

```
[Delphi]
constructor Create(const filename: string; Mode: Word);
```

```
[C++]
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode OR'ed together. The open mode must be one of the following values:

Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

Share modes

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened (VCL applications only).
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
fmShareDenyRead	Other applications can open the file for writing but not for reading (VCL applications only).
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

Shared modes available for each open mode

Open Mode	fmShareCompat (VCL)	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead (VCL)	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the *SysUtils* unit.

Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. On Windows, *Handle* is a Windows file handle. On Linux versions of CLX, it is a Linux file handle. *Handle* is read-only and reflects the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

Manipulating Files

Several common file operations are built into the runtime library. The routines for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead.

Warning: Although the Delphi language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

The following topics describe how to use runtime library routines to perform file manipulation tasks:

- Deleting a file
- Finding a file
- Renaming a file
- File date-time routines
- Copying a file

Deleting a File

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm before deleting files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

DeleteFile returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

Finding a File

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminate a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file.

```
[Delphi]
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;//Time contains the time stamp of the file.
    Size: Integer;//Size contains the size of the file in bytes.
    Attr: Integer;//Attr represents the file attributes of the file.
    Name: TFileName;//Name contains the filename and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;//FindData contains additional information such as
    //file creation time, last access time, long and short filenames.
  end;
```

```
[C++]
struct TSearchRec
{
int Time; // time stamp of the file
int Size; // size of the file in bytes
int Attr; // file attribute flags
AnsiString Name; // filename and extension
int ExcludeAttr; // file attribute flags for files to ignore
unsigned FindHandle;
_WIN32_FIND_DATAA FindData; // structure with addition information
} ;
```

On field of *TSearchRec* that is of particular interest is the *Attr* field. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Attribute constants and values

Constant	Value	Description
faReadOnly	$00000001	Read-only files
faHidden	$00000002	Hidden files
faSysFile	$00000004	System files
faVolumeID	$00000008	Volume ID files
faDirectory	$00000010	Directory files
faArchive	$00000020	Archive files
faAnyFile	$0000003F	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant using the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*:

```
[Delphi]
(SearchRec.Attr and faHidden > 0).
```

```
[C++]
(SearchRec.Attr & faHidden > 0).
```

Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass the following as the *Attr* parameter.

```
[Delphi]
(faReadOnly or faHidden).
```

```
[C++]
(faReadOnly | faHidden).
```

The following example illustrates the use of the three file find routines. It uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label:

```
[Delphi]
var
  SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\MyProgram\bin\

*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + " bytes in size';
end;
procedure TForm1.AgainClick(Sender: TObject);
begin
  if FindNext(SearchRec) = 0 then
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size'
  else
    FindClose(SearchRec);
end;
```

```
[C++]
  TSearchRec SearchRec; // global variable
void __fastcall TForm1::SearchClick(TObject *Sender)
{
  FindFirst("c:\\Program Files\\MyProgram\\bin\\
*.*", faAnyFile, SearchRec);
  Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
}
void __fastcall TForm1::AgainClick(TObject *Sender)
{
  if (FindNext(SearchRec) == 0)
    Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
  else
    FindClose(SearchRec);
}
```

```
[Delphi]
```

Note: In cross-platform applications, you should replace any hard-coded pathnames with the correct pathname for the system or use environment variables (on the Environment Variables page when you choose **Tools** ▸ **Options** ▸ **Environment Options**) to represent them.

Renaming a File

To change a file name, use the *RenameFile* function:

```
[Delphi]
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

```
[C++]
extern PACKAGE bool __fastcall RenameFile(const AnsiString OldName, const AnsiString
NewName);
```

RenameFile changes a file name, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file (for example, if a file called *NewFileName* already exists), *RenameFile* returns *False*. For example:

```
[Delphi]
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

```
[C++]
if (!RenameFile("OLDNAME.TXT","NEWNAME.TXT"))
  ErrorMsg("Error renaming file!");
```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

Note: *RenameFile* in the runtime library is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

File Date-time Routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or –1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, use a *Handle* type as a parameter. To get the file handle either:

- Use the *FileOpen* or *FileCreate* function to create a new file or open an existing file. Both *FileOpen* and *FileCreate* return the file handle.
- Instantiate *TFileStream* to create or open a file. Then use its *Handle* property. See Using file streams for more information.

Copying a File

The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the runtime library file routines, *CopyFile* takes a filename as a parameter, not a file handle. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the *RenameFile* function nor the Windows API *MoveFile* function can rename or move files across drives.

Working with ini Files and the System Registry

Many applications use ini files to store configuration information. The VCL/RTL includes two classes for working with ini files: *TIniFile* and *TMemIniFile*. Using ini files has the advantage that they can be used in cross-platform applications and they are easy to read and edit. For information on these classes, see Using TIniFile and TMemIniFile for more information.

Many Windows applications replace the use of ini files with the system Registry. The Windows system Registry is a hierarchical database that acts as a centralized storage space for configuration information. The VCL includes classes for working with the system Registry. Two of these classes, *TRegistryIniFile* and *TRegistry,* are discussed here because of their similarity to the classes for working with ini files.

TRegistryIniFile is useful for cross-platform applications, because it shares a common ancestor (*TCustomIniFile*) with the classes that work with ini files. If you confine yourself to the methods of the common ancestor (*TCustomIniFile*) your application can work on both applications with a minimum of conditional code. *TRegistryIniFile* is discussed in Using TRegistryIniFile.

For applications that are not cross-platform, you can use the *TRegistry* class. The properties and methods of *TRegistry* have names that correspond more directly to the way the system Registry is organized, because it does not need to be compatible with the classes for ini files. *TRegistry* is discussed in Using TRegistry.

Using TIniFile and TMemIniFile

The ini file format is still popular, many configuration files (such as the DSK Desktop settings file) are in this format. This format is especially useful in cross-platform applications, where you can't always count on a system Registry for storing configuration information. The VCL/RTL provides two classes, *TIniFile* and *TMemIniFile*, to make reading and writing ini files very easy.

TIniFile works directly with the ini file on disk while *TMemIniFile* buffers all changes in memory and does not write them to disk until you call the *UpdateFile* method.

When you instantiate the *TIniFile* or *TMemIniFile* object, you pass the name of the ini file as a parameter to the constructor. If the file does not exist, it is automatically created. You are then free to read values using the various read methods, such as *ReadString*, *ReadDate*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the ini file, you can use the *ReadSection* method. Similarly, you can write values using methods such as *WriteBool*, *WriteInteger*, *WriteDate*, or *WriteString*.

Following is an example of reading configuration information from an ini file in a form's *OnCreate* event handler and writing values in the *OnClose* event handler.

```
[Delphi]
procedure TForm1.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Top     := Ini.ReadInteger( 'Form', 'Top', 100 );
 Left    := Ini.ReadInteger( 'Form', 'Left', 100 );
 Caption := Ini.ReadString( 'Form', 'Caption', 'New Form' );
    if Ini.ReadBool( 'Form', 'InitMax', false ) then
      WindowState = wsMaximized
    else
      WindowState = wsNormal;
  finally
    TIniFile.Free;
  end;
end;
procedure TForm1.FormClose(Sender: TObject; var Action TCloseAction)
```

```
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Ini.WriteInteger( 'Form', 'Top', Top);
    Ini.WriteInteger( 'Form', 'Left', Left);
 Ini.WriteString( 'Form', 'Caption', Caption );
    Ini.WriteBool( 'Form', 'InitMax', WindowState = wsMaximized );
  finally
    TIniFile.Free;
  end;
end;
```

```
[C++]
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
TIniFile *ini;
ini = new TIniFile( ChangeFileExt( Application->ExeName, ".INI" ) );
Top     =  ini->ReadInteger( "Form", "Top", 100 );
Left    =  ini->ReadInteger( "Form", "Left", 100 );
Caption =  ini->ReadString( "Form", "Caption",
"Default Caption" );
ini->ReadBool( "Form", "InitMax", false ) ?
WindowState = wsMaximized :
WindowState = wsNormal;
delete ini;
}
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
TIniFile *ini;
ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   ini->WriteInteger( "Form", "Top", Top );
ini->WriteInteger( "Form", "Left", Left );
ini->WriteString ( "Form", "Caption", Caption );
ini->WriteBool   ( "Form", "InitMax",
WindowState == wsMaximized );
delete ini;
}
```

Each of the Read routines takes three parameters. The first parameter identifies the section of the ini file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the ini file. Just as the Read methods gracefully handle the case when a section or value does not exist, the Write routines create the section and/or value if they do not exist. The example code creates an ini file the first time it is run that looks like this:

```
[Form]
Top=100
Left=100
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the ini values are read in when the form is created and written back out in the *OnClose* event.

Using TRegistryIniFile

Many 32-bit Windows applications store their information in the system Registry instead of ini files because the Registry is hierarchical and doesn't suffer from the size limitations of ini files. If you are accustomed to using ini files and want to move your configuration information to the Registry instead, you can use the *TRegistryIniFile* class. You may also want to use *TRegistryIniFile* in cross-platform applications if you want to use the system Registry on Windows and an ini file on Linux. You can write most of your application so that it uses the *TCustomIniFile* type. You need only conditionalize the code that creates an instance of *TRegistryIniFile* (on Windows) or *TMemIniFile* (on Linux) and assigns it to the *TCustomIniFile* your application uses.

TRegistryIniFile makes Registry entries look like ini file entries. All the methods from *TIniFile* and *TMemIniFile* (read and write) exist in *TRegistryIniFile*.

When you construct a *TRegistryIniFile* object, the parameter you pass to the constructor (corresponding to the filename for an *IniFile* or *TMemIniFile* object) becomes a key value under the user key in the registry. All sections and values branch from that root. *TRegistryIniFile* simplifies the Registry interface considerably, so you may want to use it instead of the *TRegistry* component even if you aren't porting existing code or writing a cross-platform application.

Using TRegistry

If you are writing a Windows-only application and are comfortable with the structure of the system Registry, you can use *TRegistry*. Unlike *TRegistryIniFile*, which uses the same properties and methods of other ini file components, the properties and methods of *TRegistry* correspond more directly to the structure of the system Registry. You can specify both the root key and subkey using *TRegistry*, while *TRegistryIniFile* uses HKEY_CURRENT_USER as the root key.

In addition to methods for opening, closing, saving, moving, copying, and deleting keys, *TRegistry* lets you specify the access level you want to use.

Note: *TRegistry* is not available for cross-platform programming.

The following example retrieves a value from a registry entry:

```
[Delphi]
function GetRegistryValue(KeyName: string): string;
var
  Registry: TRegistry;
begin
Registry := TRegistry.Create(KEY_READ);
  try
    Registry.RootKey = HKEY_LOCAL_MACHINE;
// False because we do not want to create it if it doesn't exist
    Registry.OpenKey(KeyName, False);
    Result := Registry.ReadString('VALUE1');
  finally
    Registry.Free;
  end;
end;
```

```
[C++]
#include <Registry.hpp>
AnsiString GetRegistryValue(AnsiString KeyName)
{
AnsiString S;
TRegistry *Registry = new TRegistry(KEY_READ);
try
{
```

```
Registry->RootKey = HKEY_LOCAL_MACHINE;
// False because we do not want to create it if it doesn't exist
Registry->OpenKey(KeyName,false);
S = Registry->ReadString("VALUE1");
}
__finally
{
delete Registry;
}
return S;
}
```

Working with Lists

The VCL/RTL includes many classes that represents lists or collections of items. They vary depending on the types of items they contain, what operations they support, and whether they are persistent.

The following table lists various list classes, and indicates the types of items they contain:

Object	Maintains
TList	A list of pointers
TThreadList	A thread-safe list of pointers
TBucketList	A hashed list of pointers
TObjectBucketList	A hashed list of object instances
TObjectList	A memory-managed list of object instances
TComponentList	A memory-managed list of components (that is, instances of classes descended from *TComponent*)
TClassList	A list of class references
TInterfaceList	A list of interface pointers.
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TCollection	Base class for many specialized classes of typed items.
TStringList	A list of strings
THashedStringList	A list of strings with the form Name=Value, hashed for performance.

Common List Operations

Although the various list classes contain different types of items and have different ancestries, most of them share a common set of methods for adding, deleting, rearranging, and accessing the items in the list.

Adding list items

Most list classes have an *Add* method, which lets you add an item to the end of the list (if it is not sorted) or to its appropriate position (if the list is sorted). Typically, the *Add* method takes as a parameter the item you are adding to the list and returns the position in the list where the item was added. In the case of bucket lists (*TBucketList* and *TObjectBucketList*), *Add* takes not only the item to add, but also a datum you can associate with that item. In the

case of collections, *Add* takes no parameters, but creates a new item that it adds. The *Add* method on collections returns the item it added, so that you can assign values to the new item's properties.

Some list classes have an *Insert* method in addition to the *Add* method. Insert works the same way as the *Add* method, but has an additional parameter that lets you specify the position in the list where you want the new item to appear. If a class has an Add method, it also has an Insert method unless the position of items is predetermined For example, you can't use *Insert* with sorted lists because items must go in sort order, and you can't use *Insert* with bucket lists because the hash algorithm determines the item position.

The only classes that do not have an *Add* method are the ordered lists. Ordered lists are queues and stacks. To add items to an ordered list, use the *Push* method instead. *Push*, like *Add*, takes an item as a parameter and inserts it in the correct position.

Deleting list items

To delete a single item from one of the list classes, use either the *Delete* method or the *Remove* method. *Delete* takes a single parameter, the index of the item to remove. *Remove* also takes a single parameter, but that parameter is a reference to the item to remove, rather than its index. Some list classes support only a *Delete* method, some support only a *Remove* method, and some have both.

As with adding items, ordered lists behave differently than all other lists. Instead of using a *Delete* or *Remove* method, you remove an item from an ordered list by calling its *Pop* method. *Pop* takes no arguments, because there is only one item that can be removed.

If you want to delete all of the items in the list, you can call the *Clear* method. Clear is available for all lists except ordered lists.

Accessing list items

All list classes (except *TThreadList* and the ordered lists)have a property that lets you access the items in the list. Typically, this property is called *Items*. For string lists, the property is called *Strings*, and for bucket lists it is called *Data*. The *Items*, *Strings*, or *Data* property is an indexed property, so that you can specify which item you want to access.

On *TThreadList*, you must lock the list before you can access items. When you lock the list, the *LockList* method returns a *TList* object that you can use to access the items.

Ordered lists only let you access the "top" item of the list. You can obtain a reference to this item by calling the *Peek* method.

Rearranging list items

Some list classes have methods that let you rearrange the items in the list. Some have an *Exchange* method, that swaps the position of two items. Some have a *Move* method that lets you move an item to a specified location. Some have a *Sort* method that lets you sort the items in the list.

To see what methods are available, check the online Help for the list class you are using.

Persistent Lists

Persistent lists can be saved to a form file. Because of this, they are often used as the type of a published property on a component. You can add items to the list at design time, and those items are saved with the object so that they are there when the component that uses them is loaded into memory at runtime. There are two main types of persistent lists: string lists and collections.

Examples of string lists include *TStringList* and *THashedStringList*. String lists, as the name implies, contain strings. They provide special support for strings of the form Name=Value, so that you can look up the value associated with

a name. In addition, most string lists let you associate an object with each string in the list. String lists are described in more detail in Working with string lists.

Collections descend from the class *TCollection*. Each *TCollection* descendant is specialized to manage a specific class of items, where that class descends from *TCollectionItem*. Collections support many of the common list operations. All collections are designed to be the type of a published property, and many can not function independently of the object that uses them to implement on of its properties. At design time, the property whose value is a collection can use the collection editor to let you add, remove, and rearrange items. The collection editor provides a common user interface for manipulating collections.

Working with String Lists

One of the most commonly used types of list is a list of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL/RTL provides a common interface to any list of strings through an object called *TStrings* and its descendants such as *TStringList* and *THashedStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are a *TStrings* descendant) and then use these lines as items in a combo box (also a *TStrings* descendant).

A string-list property appears in the **Object Inspector** with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list

Loading and Saving String Lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the MyFile.ini file into a memo field and makes a backup copy called MyFile.bak.

```
[Delphi]
procedure EditWinIni;
var  FileName: string;{ storage for file name }
begin  FileName := 'c:\Program Files\MyProgram\MyFile.ini'{ set the file name }


  with Form1.Memo1.Lines do   begin
    LoadFromFile(FileName);{ load from file }
```

```
    SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
  end;
end;
```

```
[C++]
void __fastcall EditWinIni()
{
  AnsiString FileName = "C:\\Program Files\\MyFile.ini";
  Form1->Memo1->Lines->LoadFromFile(FileName);      // load from file
```

```
[Delphi]
```

Creating a New String List

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try... finally** block to ensure that the memory is freed even if an exception occurs.

To create a short-term string list:

1. Construct the string-list object.
2. In the try part of a **try... finally** block, use the string list.
3. In the finally part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try    { use the string list }
  finally    TempList.Free;{ destroy the list object }
  end;
end;
```

```
[C++]
void __fastcall TForm1::ButtonClick1(TObject *Sender)
{
   TStringList *TempList = new TStringList; // declare the list
   try{
      //use the string list
   }
```

```
    __finally{
delete TempList; // destroy the list object
    }
}
```

Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

To create a long-term string list:

1. In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.
2. Write an event handler for the main form's OnCreate event that executes before the form appears. It should create a string list and assign it to the field you declared in the first step.
3. Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```
[Delphi]
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
{For CLX apps: uses SysUtils, Variants, Classes, QGraphics, QControls, QForms, QDialogs;}
type  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X,
Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;
var  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
        ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.log'));{ save the list }
 ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  Shift: TShiftState;
X, Y: Integer);
begin
    ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;
```

```
end.
```

```
[C++]
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
ClickList = new TStringList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
ClickList->SaveToFile(ChangeFileExt(Application->ExeName, ".log"));//Save the list
delete ClickList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
TShiftState Shift, int X, int Y)
{
TVarRec v[] = {X,Y};
ClickList->Add(Format("Click at (%d, %d)",v,ARRAYSIZE(v) - 1));//add a string to the list
```

Note: Although you can use events such as OnCreate and OnDestroy to allocate and free classes, using the constructor and destructor for a class is generally safer coding practice.

Manipulating Strings in a List

Operations commonly performed on string lists include:

- Counting the strings in a list
- Accessing a particular string
- Finding the position of a string in the list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

Counting the Strings in a List

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

Accessing a Particular String

The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
[Delphi]
StringList1.Strings[0] := 'This is the first string.';
```

```
[C++]
StringList1->Strings[0] = "This is the first string.";
```

is equivalent to

```
[Delphi]
StringList1[0] := 'This is the first string.';
```

```
[C++]
(*StringList1)[0] = "This is the first string.";
```

Locating Items in a String List

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the Items of a list box:

```
[Delphi]
if FileListBox1.Items.IndexOf('TargetFileName') > -1 ...
```

```
[C++]
if (FileListBox1->Items->IndexOf("TargetFileName") > -1) ...
```

Iterating Through Strings in a List

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* –1.

The following example converts each string in a list box to uppercase characters.

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);var  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do     ListBox1.Items[Index] := UpperCase
(ListBox1.Items[Index]);
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int i = 0; i < ListBox1->Items->Count; i++)
```

```
    ListBox1->Items->Strings[i] = UpperCase(ListBox1->Items->Strings[i]);
}
```

Adding a String to a List

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string "Three" the third string in a list, you would use:

```
[Delphi]
Insert(2, 'Three');
```

```
[C++]
StringList1->Insert(2, "Three");
```

To append the strings from one list onto another, call *AddStrings*:

```
[Delphi]
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to StringList1 }
```

```
[C++]
StringList1->AddStrings(StringList2);  // append the strings from StringList2 to StringList1
```

Deleting a String from a List

To delete a string from a string list, call the list's *Delete* method, passing the index of the string you want to delete. If you don't know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

The following example uses *IndexOf* and *Delete* to find and delete a string:

```
[Delphi]
with ListBox1.Items do
  begin
    FoundIndex := IndexOf('bureaucracy');
    if FoundIndex > -1 then
      Delete(FoundIndex);
  end;
```

```
[C++]
int BIndex = ListBox1->Items->IndexOf("bureaucracy");
if (BIndex > -1)
  ListBox1->Items->Delete(BIndex);
```

Copying a Complete String List

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
[Delphi]
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
```

```
[C++]
Memo1->Lines->Assign(ComboBox1->Item)s;
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
[Delphi]
Memo1.Lines.AddStrings(ComboBox1.Items);   { appends strings to end }
```

```
[C++]
Memo1->Lines->AddStrings(ComboBox1->Items);
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
[Delphi]
StringList1 := StringList2;
```

```
[C++]
StringList1 = StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

Associating Objects with a String List

In addition to the strings stored in its *Strings* property, a string list can maintain references to *objects*, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear*, and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

Working with Strings

The runtime library provides many specialized string-handling routines specific to a string type. These are routines for wide strings, long strings, and null-terminated strings (meaning *PChars*). Routines that deal with null-terminated strings use the null-termination to determine the length of the string. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions. For more details about the various string types, see the *Delphi Language Guide*.

The following topics provide an overview of many of the string-handling routines in the runtime library:

- Wide character routines
- Commonly used long string routines

- Commonly used routines for null-terminated strings

Wide Character Routines

Wide strings are used in a variety of situations. Some technologies, such as XML, use wide strings as a native type. You may also choose to use wide strings because they simplify some of the string-handling issues in applications that have multiple target locales. Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second part of a character for the start of a different character.

A disadvantage of working with wide characters is that many VCL controls represent string values as single byte or MBCS strings. (Cross-platform versions of the controls typically use wide strings.) Translating between the wide character system and the MBCS system every time you set a string property or read its value can require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

The following functions convert between standard single-byte character strings (or MBCS strings) and Unicode strings:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

In addition, the following functions translate between WideStrings and other representations:

- UCS4StringToWideString
- WideStringToUCS4String
- VarToWideStr
- VarToWideStrDef

The following routines work directly with WideStrings:

- WideCompareStr
- WideCompareText
- WideSameStr
- WideSameText
- WideSameCaption (CLX applications only)
- WideFmtStr
- WideFormat
- WideLowerCase
- WideUpperCase

Finally, some routines include overloads for working with wide strings:

- UniqueString
- Length

- Trim
- TrimLeft
- TrimRight

Commonly Used Long String Routines

The long string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether they use a particular criterion in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether a routine satisfies the following criteria.

- Uses case sensitivity: If locale settings are used, it determines the definition of case. If the routine does not use locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.
- Uses locale settings: Locale settings allow you to customize your application for specific locales, in particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the system locale are typically prefaced with Ansi (that is, AnsiXXX).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented by one or more character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS parse one- and multibyte characters.

ByteType and *StrByteType* determine whether a particular byte is the lead byte of a multibyte character. Be careful when using multibyte characters not to truncate a string by cutting a character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see Enabling Application Code.

String comparison routines:

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiCompareStr	yes	yes	yes
AnsiCompareText	no	yes	yes
AnsiCompareFileName	no (yes in CLX)	yes	yes
AnsiMatchStr	yes	yes	yes
AnsiMatchText	no	yes	yes
AnsiContainsStr	yes	yes	yes
AnsiContainsText	no	yes	yes
AnsiStartsStr	yes	yes	yes
AnsiStartsText	no	yes	yes
AnsiEndsStr	yes	yes	yes

AnsiEndsText	no	yes	yes
AnsiIndexStr	yes	yes	yes
AnsiIndexText	no	yes	yes
CompareStr	yes	no	no
CompareText	no	no	no
AnsiResemblesText	no	no	no

Case conversion routines:

Routine	Uses locale settings	Supports MBCS
AnsiLowerCase	yes	yes
AnsiLowerCaseFileName	yes	yes
AnsiUpperCaseFileName	yes	yes
AnsiUpperCase	yes	yes
LowerCase	no	no
UpperCase	no	no

Note: The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the system locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

String modification routines:

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
AnsiReplaceStr	yes	yes
AnsiReplaceText	no	yes
StringReplace	optional by flag	yes
ReverseString	NA	no
StuffString	NA	no
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

Sub-string routines:

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes

LastDelimiter	yes	yes
LeftStr	NA	no
RightStr	NA	no
MidStr	NA	no
QuotedStr	no	no

Commonly Used Routines for Null-terminated Strings

The null-terminated string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string
- Copying

Where appropriate, the tables also provide columns indicating whether the routine is case-sensitive, uses the current locale, and/or supports multi-byte character sets.

Null-terminated string comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiStrComp	yes	yes	yes
AnsiStrIComp	no	yes	yes
AnsiStrLComp	yes	yes	yes
AnsiStrLIComp	no	yes	yes
StrComp	yes	no	no
StrIComp	no	no	no
StrLComp	yes	no	no
StrLIComp	no	no	no

Null-terminated case conversion routines

Routine	Uses locale settings	Supports MBCS
AnsiStrLower	yes	yes
AnsiStrUpper	yes	yes
StrLower	no	no
StrUpper	no	no

String modification routines

Routine
StrCat

StrLCat

Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiStrPos	yes	yes
AnsiStrScan	yes	yes
AnsiStrRScan	yes	yes
StrPos	yes	no
StrScan	yes	no
StrRScan	yes	no

Null-terminated string copying

Routine
StrCopy
StrLCopy
StrECopy
StrMove
StrPCopy
StrPLCopy

Declaring and Initializing Strings

When you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
  S = EmptyStr;
```

or test against an empty string:

```
  S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i];    // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string;   // empty string
begin
  proc(PChar(S));  // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
  S := 'No longer nil';
  proc(PChar(S));// proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
  SetLength(S, 100);//sets the dynamic length of S to 100
  proc(PChar(S));// proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, S is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a **string** that:

```
  S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
  S: string;
  SetLength(S, n);
```

Mixing and Converting String Types

Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

Additional functions (*CopyQStringListToTstrings, Copy TStringsToQStringList, QStringListToTStringList*) are provided for converting underlying Qt string types and CLX string types. These functions are located in Qtypes.pas.

String to PChar Conversions

Long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.

- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in the following topics:

- String dependencies
- Returning a PChar local variable
- Passing a local variable as a PChar

String Dependencies

Sometimes you need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

Returning a PChar Local Variable

A common error when working with *PChars* is to store a local variable in a data structure, or return it as a value. When your routine ends, the *PChar* disappears because it is a pointer to memory, and not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

Passing a Local Variable as a PChar

Consider the case where you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
```

```
  . . .
end;
// assume MAX_SIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, buf, SizeOf(buf));// treats buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. The *Length* of the string is automatically set to the right value after assigning *buf* to the string.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE;// when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;
```

Compiler Directives for Strings

The following compiler directives affect character and string types.

Compiler directives for strings

Directive	Description
{$H+/-}	A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive.
{$P+/-}	The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.

{$V+/-}	The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.
	
	In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:
	
	`var S: string[3];`
	`procedure Test(var T: string);`
	`begin`
	`  T := '1234';`
	`end;`

```
begin
  Test(S);
end.
```

{$X+/-}	The {$X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in PChar type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the System unit.)

Creating Drawing Spaces

The *TCanvas* class is defined in the Graphics unit, and encapsulates a Windows device context. This class handles all drawing for forms, visual containers (such as panels) and the printer object (see Printing). Using the canvas object, you need not worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

TCanvas includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw text onto the form:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
begin
Canvas.Pen.Color := clBlue;
Canvas.MoveTo( 10, 10 );
Canvas.LineTo( 100, 100 );
Canvas.Brush.Color := clBtnFace;
Canvas.Font.Name := 'Arial';
Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'This is the end of the line' );
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Canvas->Pen->Color = clBlue;
Canvas->MoveTo( 10, 10 );
Canvas->LineTo( 100, 100 );
Canvas->Brush->Color = clBtnFace;
Canvas->Font->Name = "Arial";
Canvas->TextOut( Canvas->PenPos.x, Canvas->PenPos.y,"This is the end of the line" );
}
```

The *TCanvas* object defined in the Graphics unit also protects you against common Windows graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. *TCanvas* is used everywhere in the VCL that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

Printing

The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add Printers to your **uses** clause.

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
r: TRect;
i: Integer;
begin
with Printer do
begin
r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
BeginDoc;
      Canvas.Brush.Style := bsClear;
for i := 0 to Memo1.Lines.Count do
Canvas.TextOut(200,200 + (i *
                  Canvas.TextHeight(Memo1.Lines.Strings[i])),
                  Memo1.Lines.Strings[i]);
Canvas.Brush.Color := clBlack;
Canvas.FrameRect(r);
EndDoc;
end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TPrinter *Prntr = Printer();
TRect r = Rect(200,200,Prntr->PageWidth - 200,Prntr->PageHeight- 200);
Prntr->BeginDoc();
for( int i = 0; i < Memo1->Lines->Count; i++)
Prntr->Canvas->TextOut(200,200 + (i *
  Prntr->Canvas->TextHeight(Memo1->Lines->Strings[i])),
  Memo1->Lines->Strings[i]);
Prntr->Canvas->Brush->Color = clBlack;
Prntr->Canvas->FrameRect(r);
Prntr->EndDoc();
}
```

Converting Measurements

The ConvUtils unit declares a general-purpose Conversion Function that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see Performing Conversions.

The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the RegisterConversionTypeand RegisterConversionFamily functions. For information on extending conversion and conversion units, see Adding new measurement types.

Performing Conversions

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the TConvType type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
[Delphi]
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

```
[C++]
TempInKelvin = Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
[Delphi]
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

```
[C++]
double nKPL = Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling CompatibleConversionTypes.

The StdConvs unit defines several families of *TConvType* values.

Adding New Measurement Types

If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the RegisterConversionFamily function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit), you can use the RegisterConversionType function to add the new units to the conversion family. The following examples show how to do this:

Creating a simple conversion family and adding units

Using a conversion function

Using a class to manage conversions

For more examples, refer to the source code for the standard conversions unit (stdconvs.pas). (Note that the source is not included in all editions of Delphi.)

Creating a Simple Conversion Family and Adding Units

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new LongTime conversion family, and the units of measurement that are its members:

```
[Delphi]
var
cbLongTime: TConvFamily;
ltMonths: TConvType;
ltYears: TConvType;
ltDecades: TConvType;
ltCenturies: TConvType;
ltMillennia: TConvType;
```

```
[C++]
tConvFamily cbLongTime;
TConvType ltMonths;
TConvType ltYears;
TConvType ltDecades;
TConvType ltCenturies;
TConvType ltMillennia;
```

Register the conversion family

Next, register the conversion family:

```
[Delphi]
cbLongTime := RegisterConversionFamily ('Long Times');
```

```
[C++]
cbLongTime = RegisterConversionFamily ("Long Times");
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

```
[Delphi]
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);
ltYears:=RegisterConversionType(cbLongTime,'Years',1);
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
```

```
[C++]
ltMonths = RegisterConversionType(cbLongTime,"Months",1/12);
ltYears = RegisterConversionType(cbLongTime,"Years",1);
ltDecades = RegisterConversionType(cbLongTime,"Decades",10);
ltCenturies = RegisterConversionType(cbLongTime,"Centuries",100);
ltMillennia = RegisterConversionType(cbLongTime,"Millennia",1000);
```

Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
[Delphi]
Convert(StrToFloat(Edit1.Text),tuMonths,tuMillennia);
```

```
[C++]
Convert(StrToFloat(Edit1->Text),tuMonths,tuMillennia);
```

you can now use this one for greater accuracy:

```
[Delphi]
Convert(StrToFloat(Edit1.Text),ltMonths,ltMillennia);
```

```
[C++]
Convert(StrToFloat(Edit1->Text),ltMonths,ltMillennia);
```

Using a Conversion Function

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units.

Declare variables

First, declare variables for the identifiers. The identifiers are used in the *cbTemperature* conversion family, and the units of measurement are its members:

```
[Delphi]
var
cbTemperature: TConvFamily;
tuCelsius: TConvType;
tuKelvin: TConvType;
tuFahrenheit: TConvType;
```

```
[C++]
TConvFamily cbTemperature;
TConvType tuCelsius;
TConvType tuKelvin;
TConvType tuFahrenheit;
```

Note: The units of measurement listed here are a subset of the temperature units actually registered in the *StdConvs* unit.

Register the conversion family

Next, register the conversion family:

```
[Delphi]
cbTemperature := RegisterConversionFamily ('Temperature');
```

```
[C++]
cbTemperature = RegisterConversionFamily ("Temperature");
```

Register the base unit

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
[Delphi]
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
```

```
[C++]
tuCelsius = RegisterConversionType(cbTemperature,"Celsius",1);
```

Write methods to convert to and from the base unit

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the StdConvs unit:

```
[Delphi]
function FahrenheitToCelsius(const AValue: Double): Double;
begin
Result := ((AValue - 32) * 5) / 9;
end;
function CelsiusToFahrenheit(const AValue: Double): Double;
begin
Result := ((AValue * 9) / 5) + 32;
end;
function KelvinToCelsius(const AValue: Double): Double;
begin
Result := AValue - 273.15;
end;
function CelsiusToKelvin(const AValue: Double): Double;
begin
Result := AValue + 273.15;
end;
```

```
[C++]
double __fastcall FahrenheitToCelsius(const double AValue)
{
return (((AValue - 32) * 5) / 9);
}
double __fastcall CelsiusToFahrenheit(const double AValue)
{
return (((AValue * 9) / 5) + 32);
}
double __fastcall KelvinToCelsius(const double AValue)
{
  return (AValue - 273.15);
}
double __fastcall CelsiusToKelvin(const double AValue)
{
  return (AValue + 273.15);
}
```

Register the other units

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also include a description of the units.

The code to register the other units in the family is shown here:

```
[Delphi]
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
CelsiusToKelvin);
tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
```

```
[C++]
tuKelvin = RegisterConversionType(cbTemperature, "Kelvin", KelvinToCelsius,
CelsiusToKelvin);
tuFahrenheit = RegisterConversionType(cbTemperature, "Fahrenheit", FahrenheitToCelsius,
CelsiusToFahrenheit);
```

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
[Delphi]
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

```
[C++]
Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

Using a Class to Manage Conversions

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

```
[Delphi]
function FromEuro(const AValue: Double, Factor; FRound: TRoundToRange): Double;
begin
Result := RoundTo(AValue * Factor, FRound);
end;
function ToEuro(const AValue: Double, Factor): Double;
begin
Result := AValue / Factor;
end;
```

```
[C++]
double __fastcall FromEuro(const double AValue, const double Factor, TRoundToRange FRound)
{
return(RoundTo(AValue * Factor, FRound));
}
double __fastcall ToEuro(const double AValue, const double Factor)
{
return (AValue / Factor);
}
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class:

```
[Delphi]
type
TConvTypeEuroFactor = class(TConvTypeFactor)
private
FRound: TRoundToRange;
public
constructor Create(const AConvFamily: TConvFamily;
const ADescription: string; const AFactor: Double;
const ARound: TRoundToRange);
function ToCommon(const AValue: Double): Double; override;
function FromCommon(const AValue: Double): Double; override;
end;
end;
```

```
[C++]
class PASCALIMPLEMENTATION TConvTypeEuroFactor : public Convutils::TConvTypeFactor
{
  private:
TRoundToRange FRound;
public:
    __fastcall TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound);
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
    virtual double ToCommon(const double AValue);
virtual double FromCommon(const double AValue);
}
```

The constructor assigns values to those private members:

```
[Delphi]
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
const ADescription: string; const AFactor: Double;
const ARound: TRoundToRange);
```

```
begin
inherited Create(AConvFamily, ADescription, AFactor);
FRound := ARound;
end;
```

```
[C++]
__fastcall TConvTypeEuroFactor::TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound):
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
{
  FRound = ARound;
}
```

The two conversion functions simply use these private members:

```
[Delphi]
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
Result := RoundTo(AValue * Factor, FRound);
end;
function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
Result := AValue / Factor;
end;
```

```
[C++]
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (RoundTo(AValue * Factor, FRound));
}
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (AValue / Factor);
}
```

Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

```
[Delphi]
var
  euEUR: TConvType; { EU euro }
euBEF: TConvType; { Belgian francs }
euDEM: TConvType; { German marks }
euGRD: TConvType; { Greek drachmas }
euESP: TConvType; { Spanish pesetas }
euFFR: TConvType; { French francs }
euIEP: TConvType; { Irish pounds }
euITL: TConvType; { Italian lire }
euLUF: TConvType; { Luxembourg francs }
euNLG: TConvType; { Dutch guilders }
euATS: TConvType; { Austrian schillings }
euPTE: TConvType; { Portuguese escudos }
euFIM: TConvType; { Finnish marks }
  cbEuro: TConvFamily;
```

```
[C++]
TConvFamily cbEuro;
TConvType euEUR; // EU euro
TConvType euBEF; // Belgian francs
TConvType euDEM; // German marks
TConvType euGRD; // Greek drachmas
TConvType euESP; // Spanish pesetas
TConvType euFFR; // French francs
TConvType euIEP; // Irish pounds
TConvType euITL; // Italian lire
TConvType euLUF; // Luxembourg francs
TConvType euNLG; // Dutch guilders
TConvType euATS; // Austrian schillings
TConvType euPTE; // Protuguese escudos
TConvType euFIM; // Finnish marks
```

Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class. Register the conversion family the same way you registered the other conversion families:

```
[Delphi]
cbEuro := RegisterConversionFamily ('European currency');
```

```
[C++]
cbEuro = RegisterConversionFamily ("European currency");
```

To register each conversion type, create an instance of the conversion class that reflects the factor and rounding properties of that currency, and call the RegisterConversionType method:

```
[Delphi]
var
  LInfo: TConvTypeInfo;
begin
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'EUEuro', 1.0, -2);
  if not RegisterConversionType(LInfo, euEUR) then
    LInfo.Free;
```

```
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'BelgianFrancs', 40.3399, 0);
  if not RegisterConversionType(LInfo, euBEF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GermanMarks', 1.95583, -2);
  if not RegisterConversionType(LInfo, euDEM) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GreekDrachmas', 340.75, 0);
  if not RegisterConversionType(LInfo, euGRD) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'SpanishPesetas', 166.386, 0);
  if not RegisterConversionType(LInfo, euESP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FrenchFrancs', 6.55957, -2);
  if not RegisterConversionType(LInfo, euFFR) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'IrishPounds', 0.787564, -2);
  if not RegisterConversionType(LInfo, euIEP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'ItalianLire', 1936.27, 0);
  if not RegisterConversionType(LInfo, euITL) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'LuxembourgFrancs', 40.3399, -2);
  if not RegisterConversionType(LInfo, euLUF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'DutchGuilders', 2.20371, -2);
  if not RegisterConversionType(LInfo, euNLG) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'AustrianSchillings', 13.7603, -2);
  if not RegisterConversionType(LInfo, euATS) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'PortugueseEscudos', 200.482, -2);
  if not RegisterConversionType(LInfo, euPTE) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FinnishMarks', 5.94573, 0);
  if not RegisterConversionType(LInfo, euFIM) then
    LInfo.Free;
end;
```

```
[C++]
TConvTypeInfo *pInfo = new TConvTypeEuroFactor(cbEuro, "EUEuro", 1.0, -2);
if (!RegisterConversionType(pInfo, euEUR))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "BelgianFrancs", 40.3399, 0);
if (!RegisterConversionType(pInfo, euBEF))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GermanMarks", 1.95583, -2);
if (!RegisterConversionType(pInfo, euDEM))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GreekDrachmas", 340.75, 0);
if (!RegisterConversionType(pInfo, euGRD)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "SpanishPesetas", 166.386, 0);
if (!RegisterConversionType(pInfo, euESP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FrenchFrancs", 6.55957, -2);
if (!RegisterConversionType(pInfo, euFFR)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "IrishPounds", 0.787564, -2);
if (!RegisterConversionType(pInfo, euIEP)
  delete pInfo;
```

```
pInfo = new TConvTypeEuroFactor(cbEuro, "ItalianLire", 1936.27, 0);
if (!RegisterConversionType(pInfo, euITL)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "LuxembourgFrancs", 40.3399, -2);
if (!RegisterConversionType(pInfo, euLUF)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "DutchGuilders", 2.20371, -2);
if (!RegisterConversionType(pInfo, euNLG)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "AutstrianSchillings", 13.7603, -2);
if (!RegisterConversionType(pInfo, euATS)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "PortugueseEscudos", 200.482, -2);
if (!RegisterConversionType(pInfo, euPTE)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FinnishMarks", 5.94573, 0);
if (!RegisterConversionType(pInfo, euFIM)
  delete pInfo;
```

Note: The ConvertIt demo provides an expanded version of this example that includes other currencies (that do not have fixed conversion rates) and more error checking.

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new cbEuro family. For example, the following code converts a value from Italian Lire to German Marks:

```
[Delphi]
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

```
[C++]
Edit2->Text = FloatToStr(Convert(StrToFloat(Edit1->Text), euITL, euDEM));
```

Defining Custom Variants

One powerful built-in type of the Delphi language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

To create a Variant type:

1. Map the storage of the variant's data on to the *TVarData* record.
2. Declare a class that descends from TCustomVariantType. Implement all required behavior (including type conversion rules) in the new class.

3 Write utility methods for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods, you use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

Storing a Custom Variant Type's Data

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the VarConv unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The VarConv unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

Note: When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the VarCmplx unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

Creating a Class to Enable the Custom Variant Type

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of TCustomVariantType. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

The following topics provide details on how to implement and use a TCustomVariantType descendant:

- Enabling casting

- Implementing binary operations
- Implementing comparison operations
- Implementing unary operations
- Copying and clearing custom variants
- Loading and saving custom variant values
- Using the TCustomVariantType descendant

Enabling Casting

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: Cast, which converts another variant type to your custom variant, and CastTo, which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type Double as an intermediate type:

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
LSource, LTemp: TVarData;
begin
VarDataInit(LSource);
try
VarDataCopyNoInd(LSource, Source);
if VarDataIsStr(LSource) then
TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
else
begin
VarDataInit(LTemp);
try
VarDataCastTo(LTemp, LSource, varDouble);
TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
finally
VarDataClear(LTemp);
end;
end;
Dest.VType := VarType;
finally
VarDataClear(LSource);
end;
end;
```

In addition to the use of Double as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the VarType property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.

- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to store its data (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of Double (for any destination type other than a string):

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
const AVarType: TVarType);
var
LTemp: TVarData;
begin
if Source.VType = VarType then
case AVarType of
varOleStr:
VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
varString:
VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
else
VarDataInit(LTemp);
try
LTemp.VType := varDouble;
LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
VarDataCastTo(Dest, LTemp, AVarType);
finally
VarDataClear(LTemp);
end;
end
else
RaiseCastError;
end;
```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

Implementing Binary Operations

To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the BinaryOp method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
        else
          RaiseInvalidOp;
        end;
      else
```

```
        if Left.VType = VarType then
          case Operator of
            opAdd:
              TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
            opSubtract:
              TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
            opMultiply:
              TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
            opDivide:
              TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
            else
              RaiseInvalidOp;
          end
        else
          RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the RightPromotion method so that it always requires the type in the *VarType* property:

```
function TComplexVariantType.RightPromotion(const V: TVarData;
const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
{ Complex Op TypeX }
RequiredVarType := VarType;
Result := True;
end;
```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

BinaryOp only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar method, LeftPromotion, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
{ TypeX Op Complex }
if (Operator = opAdd) and VarDataIsStr(V) then
RequiredVarType := varString
else
RequiredVarType := VarType;
```

```
  Result := True;
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

Implementing Comparison Operations

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The Compare method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the VarConv unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
var Relationship: TVarCompareResult);
const
CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
(crLessThan, crEqual, crGreaterThan);
var
LValue: Double;
LType: TConvType;
LRelationship: TValueRelationship;
begin
// supports...
//   convvar cmp number
//     Compare the value of convvar and the given number
  //   convvar1 cmp convvar2
//     Compare after converting convvar2 to convvar1's unit type
  //  The right can also be a string.  If the string has unit info then it is
//    treated like a varConvert else it is treated as a double
LRelationship := EqualsValue;
case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                                              TConvertVarData(Left).VConvType, LValue, LType)
        else
          RaiseCastError;
    varDouble:
      LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
    else
      if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                          TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
                          TConvertVarData(Right).VConvType)
      else
        RaiseInvalidOp;
  end;
  Relationship := CRelationshipToRelationship[LRelationship];
end;
```

If the custom type does not support the concept of "greater than" or "less than," only "equal" or "not equal," however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or

crGreaterThan. When the only valid response is "not equal," it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

CompareOp has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is *True*. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the VarCmplx unit. It supports only a test of equality or inequality:

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;
const Operator: Integer): Boolean;
begin
Result := False;
if (Left.VType = VarType) and (Right.VType = VarType) then
case Operator of
opCmpEQ:
Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
opCmpNE:
Result := not TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
else
RaiseInvalidOp;
end
else
RaiseInvalidOp;
end;
```

Note that the types of operands that both these implementations support are very limited. As with binary operations, you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

Implementing Unary Operations

To allow the custom variant type to work with standard unary operators (-, not), you must override the UnaryOp method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
if Right.VType = VarType then
case Operator of
opNegate:
TComplexVarData(Right).VComplex.DoNegate;
else
RaiseInvalidOp;
end
else
RaiseInvalidOp;
end;
```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

Copying and Clearing Custom Variants

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant's value, implement the Copy method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
const Indirect: Boolean);
begin
if Indirect and VarDataIsByRef(Source) then
VarDataCopyNoInd(Dest, Source)
else
with TComplexVarData(Dest) do
begin
VType := VarType;
VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
end;
end;
```

Note: The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

Tip: If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the Copy method can simply call the SimplisticCopy method.

To indicate how to clear the variant's value, implement the Clear method. As with the Copy method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
V.VType := varEmpty;
FreeAndNil(TComplexVarData(V).VComplex);
end;
```

You will also need to implement the IsClear method. This way, you can detect any invalid values or special values that represent "blank" data:

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
Result := (TComplexVarData(V).VComplex = nil) or
TComplexVarData(V).VComplex.IsZero;
end;
```

Loading and Saving Custom Variant Values

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the IVarStreamable interface from Classes.pas.

IVarStreamable defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the VarCmplx unit, implements the *IVarStreamable* methods as follows:

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
with TReader.Create(Stream, 1024) do
try
with TComplexVarData(Dest) do
begin
VComplex := TComplexData.Create;
VComplex.Real := ReadFloat;
VComplex.Imaginary := ReadFloat;
end;
finally
Free;
end;
end;
procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
with TWriter.Create(Stream, 1024) do
try
with TComplexVarData(Source).VComplex do
begin
WriteFloat(Real);
WriteFloat(Imaginary);
end;
finally
Free;
end;
end;
```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

Using the TCustomVariantType Descendant

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the VarCmplx unit:

```
initialization
ComplexVariantType := TComplexVariantType.Create;
```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the VarCmplx unit:

```
finalization
FreeAndNil(ComplexVariantType);
```

Writing Utilities to Work with a Custom Variant Type

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
TComplexVarData(Result).VType := ComplexVariantType.VarType;
TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

This function does not actually exist in the VarCmplx unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a way to easily determine the type code for your custom variant type. The following function from the VarCmplx unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
Result := ComplexVariantType.VarType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the VarCmplx unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;
function VarAsComplex(const AValue: Variant): Variant;
begin
if not VarIsComplex(AValue) then
VarCast(Result, AValue, VarComplex)
else
Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the methods implemented in the TCustomVariantType descendant for casting data.

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the VarCmplx unit defines a large number of functions that implement mathematical operations on complex-valued variants.

Supporting Properties and Methods in Custom Variants

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

Using TInvokeableVariantType

To provide support for properties and methods, the class you create to enable the new custom variant type should descend from TInvokeableVariantType instead of directly from *TCustomVariantType*.

TInvokeableVariantType defines four methods:

- DoFunction
- DoProcedure
- GetProperty
- SetProperty

that you can implement to support properties and methods on your custom variant type.

For example, the VarConv unit uses *TInvokeableVariantType* as the base class for *TConvertVariantType* so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
const V: TVarData; const Name: String): Boolean;
var
LType: TConvType;
begin
// supports...
//   'Value'
//   'Type'
//   'TypeName'
//   'Family'
//   'FamilyName'
//   'As[Type]'
Result := True;
if Name = 'VALUE' then
Variant(Dest) := TConvertVarData(V).VValue
else if Name = 'TYPE' then
Variant(Dest) := TConvertVarData(V).VConvType
else if Name = 'TYPENAME' then
Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
else if Name = 'FAMILY' then
Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
else if Name = 'FAMILYNAME' then
Variant(Dest) := ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
else if System.Copy(Name, 1, 2) = 'AS' then
begin
if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
System.Copy(Name, 3, MaxInt), LType) then
VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
TConvertVarData(V).VConvType, LType), LType)
else
Result := False;
end
else
Result := False;
end;
```

The *GetProperty* method checks the *Name* parameter to determine what property is wanted. It then retrieves the information from the *TVarData* record of the Variant (*V*), and returns it as a Variant (*Dest*). Note that this method supports properties whose names are dynamically generated at runtime (As[Type]), based on the current value of the custom variant.

Similarly, the *SetProperty*, *DoFunction*, and *DoProcedure* methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

Using TPublishableVariantType

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use TPublishableVariantType as the base class for your custom variant type, then you need only implement the GetInstance method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in Storing a custom variant type's data, *TComplexVariantType* stores the data of a complex-valued variant using an instance of *TComplexData*. *TComplexData* has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. *TComplexVariantType* descends from *TPublishableVariantType*, and implements the *GetInstance* method to return the *TComplexData* object (in TypInfo.pas) that is stored in a complex-valued variant's *TVarData* record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType does the rest. It overrides the *GetProperty* and *SetProperty* methods to use the runtime type information (RTTI) of the *TComplexData* object for getting and setting property values.

Note: For *TPublishableVariantType* to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {$M+} compiler directive, or descend from *TPersistent*.

Working with components

Setting Component Properties

To set published properties at design time, you can use the **Object Inspector** and, in some cases, special property editors. To set properties at runtime, assign their values in your application source code.

For information about the properties of each component, see the online Help.

Setting Properties at Design Time

When you select a component on a form at design time, the **Object Inspector** displays its published properties and (when appropriate) allows you to edit them. Use the Tab key to toggle between the left-hand Property column and the right-hand Value column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the **Object Inspector** and choose View.

When more than one component is selected, the **Object Inspector** displays all properties—except *Name*—that are shared by the selected components. If the value for a shared property differs among the selected components, the **Object Inspector** displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Changing code-related properties, such as the name of an event handler, in the **Object Inspector** automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the **Object Inspector**.

Using Property Editors

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the **Object Inspector**. To open the property editor, double-click in the Value column, click the ellipsis mark, or type Ctrl+Enter when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

Setting Properties at Runtime

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
[Delphi]
Form1.Caption := MyString;
```

```
[C++]
Form1->Caption = MyString;
```

Calling Methods

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a draw-grid object like this:

```
[Delphi]
DrawGrid1.Repaint;
```

```
[C++]
DrawGrid1->Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form's child controls, you don't have to prepend the name of the form to the method call:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Repaint;
}
```

For more information about scope, see Scope and Qualifiers.

Working with Events and Event Handlers

Almost all the code you write is executed, directly or indirectly, in response to events. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is a Delphi procedure. The sections that follow show how to:

- Generate a new event handler.
- Generate a handler for a component's default event.
- Locate event handlers.
- Associate an event with an existing event handler.
- Associate menu events with event handlers.

- Delete event handlers.

Generating a New Event Handler

You can generate skeleton event handlers for forms and other components.

To create an event handler:

1. Select a component.
2. Click the Events tab in the **Object Inspector**. The Events page of the **Object Inspector** displays all events defined for the component.
3. Select the event you want, then double-click the Value column or press Ctrl+Enter. The Code editor opens with the cursor inside the skeleton event handler, or **begin...end** block.
4. At the cursor, type the code that you want to execute when the event occurs.

Generating a Handler for a Component's Default Event

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

Locating Event Handlers

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default:

1. In the form, select the component whose event handler you want to locate.
2. In the **Object Inspector**, click the Events tab.
3. Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor inside the skeleton event-handler.

Associating an Event with an Existing Event Handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler

1. On the form, select the component whose event you want to handle.
2. On the Events page of the **Object Inspector**, select the event to which you want to attach a handler.
3. Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The previous procedure is an easy way to reuse event handlers. Action lists and in the VCL, action bands, however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot.

Using the Sender Parameter

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

Displaying and Coding Shared Events

When components share events, you can display their shared events in the **Object Inspector**. First, select the components by holding down the Shift key and clicking on them in the Form Designer; then choose the Events tab in the **Object Inspector**. From the Value column in the **Object Inspector**, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

Associating Menu Events with Event Handlers

The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This topic explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see Creating and managing menus.

To create an event handler for a menu item:

1. Open the Menu Designer by double-clicking on a MainMenu or PopupMenu component.
2. Select a menu item in the Menu Designer. In the **Object Inspector**, make sure that a value is assigned to the item's Name property.
3. From the Menu Designer, double-click the menu item. The Code editor opens with the cursor inside the skeleton event handler, or the **begin...end** block.

4 At the cursor, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing OnClick event handler:

1 Open the Menu Designer by double-clicking a MainMenu or PopupMenu component.

2 Select a menu item in the Menu Designer. In the **Object Inspector**, make sure that a value is assigned to the item's Name property.

3 On the Events page of the **Object Inspector**, click the down arrow in the Value column next to OnClick to open a list of previously written event handlers. (The list includes only event handlers written for OnClick events on this form.) Select from the list by clicking an event handler name.

Deleting Event Handlers

When you delete a component from a form using the Form Designer, the Code editor removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the unit's **interface** section) and its implementation (in the **implementation** section). Otherwise you'll get a compiler error when you build your project.

Cross-platform and Non-cross-platform Components

The **Tool palette** contains a selection of components that handle a wide variety of programming tasks. The components are arranged in pages according to their purpose and functionality. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page. Which pages appear in the default configuration depends on the edition of the product you are running.

The following table lists typical default pages and components available for creating applications, including those that are not cross-platform. You can use all CLX components in both Windows and Linux applications. You can use some VCL-specific components in a Windows-only CLX application; however, the application is not cross-platform unless you isolate these portions of the code.

Tool palette pages

Page name	Description	Cross-platform?
ActiveX	Sample ActiveX controls; see Microsoft documentation (msdn.microsoft.com).	No
Additional	Specialized controls.	Yes, though for VCL applications only: ApplicationEvents, ValueListEditor, ColorBox, Chart, ActionManager, ActionMainMenuBar, ActionToolBar, CustomizeDlg, and StaticText. For CLX applications only: LCDNumber.
ADO	Components that provide data access through the ADO framework.	No
BDE	Components that provide data access through the Borland Database Engine.	No
COM+	Component for handling COM+ events.	No

Data Access	Components for working with database data that are not tied to any particular data access mechanism.	Yes, though for VCL applications only: XMLTransform, XMLTransformProvider, and XMLTransformClient.
Data Controls	Visual, data-aware controls.	Yes, though for VCL applications only: DBRichEdit, DBCtrlGrid, and DBChart.
dbExpress	Database controls that use dbExpress, a cross-platform, database-independent layer that provides methods for dynamic SQL processing. It defines a common interface for accessing SQL servers.	Yes
DataSnap	Components used for creating multi-tiered database applications.	No
Decision Cube	Data analysis components.	No
Dialogs	Commonly used dialog boxes.	Yes, though for VCL applications only: OpenPictureDialog, SavePictureDialog, PrintDialog, and PrinterSetupDialog.
Indy Clients Indy Servers Indy Misc Indy Intercepts Indy I/O Handlers	Cross-platform Internet components for the client and server (open source Winshoes Internet components).	Yes
InterBase	Components that provide direct access to the InterBase database.	Yes
InterBaseAdmin	Components that access InterBase Services API calls.	Yes
Internet	Components for Internet communication protocols and Web applications.	Yes
InternetExpress	Components that are simultaneously a Web server application and the client of a multi-tiered database application.	Yes
Office2K	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation).	No
IW Client Side IW Control IW Data IW Standard	Components to build Web server applications using IntraWeb.	No
Rave	Components to design visual reports.	No
Samples	Sample custom components.	No
Servers	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation).	No
Standard	Standard controls, menus.	Yes

System	Components and controls for system-level access, including timers, multimedia, and DDE (VCL applications). Components for filtering and displaying files (CLX applications).	The components are different between a VCL and CLX application.
WebServices	Components for writing applications that implement or use SOAP-based Web Services.	Yes
WebSnap	Components for building Web server applications.	Yes
Win 3.1	Old style Win 3.1 components.	No
Win32 (VCL)/Common Controls (CLX)	Common Windows controls.	In CLX applications, the Common Controls page replaces the Win32 page. VCL applications only: RichEdit, UpDown, HotKey, DataTimePicker, MonthCalendar, CoolBar, PageScroller, and ComboBoxEx. CLX applications only: TextViewer, TextBrowser, SpinEdit, and IconView.

You can add, remove, and rearrange components on the palette, and you can create component templates and frames that group several components.

For more information about the components on the **Tool palette**, see online Help. You can press F1 on the **Tool palette**, on the component itself when it is selected, after it has been dropped onto a form, or anywhere on its name in the Code editor. If a tab of the **Tool palette** is selected, the Help gives a general description for all of the components on that tab. Some of the components on the ActiveX, Servers, and Samples pages, however, are provided as examples only and are not documented.

Adding Custom Components to the Tool Palette

You can install custom components—written by yourself or third parties—on the **Tool palette** and use them in your applications. To write a custom component, see Overview of component creation. To install an existing component, see Installing component packages.

Working with controls

Implementing Drag and Drop in Controls

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.

- Starting a drag operation
- Accepting dragged items
- Dropping items
- Ending a drag operation
- Customizing drag and drop with a drag object
- Changing the drag mouse pointer

Starting a Drag Operation

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. Calling

```
[Delphi]
BeginDrag (False);
```

```
[C++]
false, -1
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```
[Delphi]
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { drag only if left button pressed }
    with Sender as TFileListBox do  { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
          BeginDrag(False);  { if so, drag it }
    end;
end;
```

```
[C++]
void __fastcall TFMForm::FileListBox1MouseDown(TObject *Sender,
{
  if (Button == mbLeft)
  {
    TFileListBox *pLB = dynamic_cast<TFileListBox *>(Sender); // cast to TFileListBox
    if (pLB && pLB->ItemAtPos(Point(X,Y), true) >= 0) // is there an item here?
      pLB->BeginDrag(false, -1);                      // if so, drag it
  }
}
```

Accepting Dragged Items

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. *Accept* changes the cursor type to an accept cursor or not.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drag. In the following VCL example, a directory tree view accepts dragged items only if they come from a file list box.

```
[Delphi]
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
end;
```

```
[C++]
void __fastcall TForm1::TreeView1DragOver(TObject *Sender, TObject *Source,
  int X, int Y, TDragState State, bool &Accept)
{
  if (Source->InheritsFrom(__classid(TFileListBox)))
}
```

Dropping Items

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components if an item is dropped.

In the following VCL example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```
[Delphi]
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

```
[C++]
void __fastcall TForm1::TreeView1DragDrop(TObject *Sender, TObject *Source,
  if (Source->InheritsFrom(__classid(TFileListBox)))
  {
    TTreeNode *pNode = TreeView1->GetNodeAt(X,Y); // pNode is drop target
    AnsiString NewFile = pNode->Text + AnsiString("//") +
      ExtractFileName(FileListBox1->FileName); // build file name for drop target
    MoveFileEx(FileListBox1->FileName.c_str(), NewFile.c_str(),
      MOVEFILE_REPLACE_EXISTING | MOVEFILE_COPY_ALLOWED); // move the file
  }
}
```

Ending a Drag Operation

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the drag was initiated. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In the following VCL example, a file list box handles an end-drag event by refreshing its file list.

```
[Delphi]
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
```

```
[C++]
void __fastcall TFMForm::FileListBox1EndDrag(TObject *Sender, TObject *Target,   int X, int
Y)
  if (Target)
    FileListBox1->Update();
};
```

Customizing Drag and Drop with a Drag Object

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* (VCL only) and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

TDragObjectEx descendants (VCL only) are freed automatically whereas descendants of *TDragObject* are not. If you have *TDragObject* descendants that you are not explicitly freeing, you can change them so they descend from *TDragObjectEx* instead to prevent memory loss.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a DLL, or between forms that are implemented using different DLLs.

Changing the Drag Mouse Pointer

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).

Implementing Drag and Dock in Controls

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.

- Making a windowed control a docking site
- Making a control a dockable child
- Controlling how child controls are docked
- Controlling how child controls are undocked
- Controlling how child controls respond to drag-and-dock operations

Note: Drag-and-dock properties are not available in CLX applications.

Making a Windowed Control a Docking Site

To make a windowed control a docking site:

1. Set the *DockSite* property to *True*.
2. If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

Note: Drag-and-dock properties are not available in CLX applications.

Making a Control a Dockable Child

To make a control a dockable child:

1. Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.
2. Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.
3. Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

Note: Drag-and-dock properties are not available in CLX applications.

Controlling How Child Controls Are Docked

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
[Delphi]
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var   InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

```
[C++]
__property TGetSiteInfoEvent OnGetSiteInfo = {read=FOnGetSiteInfo, write=FOnGetSiteInfo};
```

```
typedef void __fastcall (__closure *TGetSiteInfoEvent)(System::TObject* Sender, TControl*
DockClient, Windows::TRect &InfluenceRect, const Windows::TPoint &MousePos, bool &CanDock);
```

OnGetSiteInfo occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is initialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
[Delphi]
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer; State:
TDragState; var Accept: Boolean) of object;
```

```
[C++]
__property TDockOverEvent OnDockOver = {read=FOnDockOver, write=FOnDockOver};
typedef void __fastcall (__closure *TDockOverEvent)(System::TObject* Sender,
TDragDockObject* Source, int X, int Y, TDragState State, bool &Accept);
```

OnDockOver occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept* parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
[Delphi]
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer) of
object;
```

```
[C++]
__property TDockDropEvent OnDockDrop = {read=FOnDockDrop, write=FOnDockDrop};
typedef void __fastcall (__closure *TDockDropEvent)(System::TObject* Sender,
TDragDockObject* Source, int X, int Y);
```

OnDockDrop occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

Note: Drag-and-dock properties are not available in CLX applications.

Controlling How Child Controls Are Undocked

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
[Delphi]
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)   of object;
```

```
[C++]
__property TUnDockEvent OnUnDock = {read=FOnUnDock, write=FOnUnDock};
typedef void __fastcall (__closure *TUnDockEvent)(System::TObject* Sender, TControl*
Client, TWinControl* NewTarget, bool &Allow);
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

Note: Drag-and-dock properties are not available in CLX applications.

Controlling How Child Controls Respond to Drag-and-dock Operations

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

Note: Drag-and-dock properties are not available in CLX applications.

Working with Text in Controls

The following topics how to use various features of rich edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scrollbars at runtime
- Adding the clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the OnPopup event

Setting Text Alignment

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to.

For example, the following code attaches an *OnClick* event handler to a **Character** ▸ **Left** menu item, then attaches the same event handler to both a **Character** ▸ **Right** and **Character** ▸ **Center** menu item.

```
[Delphi]
procedure TForm.AlignClick(Sender: TObject);
begin
```

```
  Left1.Checked := False;  { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

```
[C++]
switch(reinterpret_cast<int>(RichEdit1->Paragraph->Alignment))
{
  case 0: LeftAlign->Down   = true; break;
  case 1: RightAlign->Down  = true; break;
  case 2: CenterAlign->Down = true; break;
}
```

You can also use the *HMargin* property to adjust the left and right margins in a memo control.

Adding Scroll Bars at Runtime

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime:

1. Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.
2. Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a **Character** ▸ **WordWrap** menu item.

```
[Delphi]
procedure TForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
  end;
end;
```

```
[C++]
void __fastcall TForm::WordWrap1Click(TObject *Sender)
{
  Editor->WordWrap = !(Editor->WordWrap);   // toggle word wrapping
```

```
  if (Editor->WordWrap)
    Editor->ScrollBars = ssVertical;        // wrapped requires only vertical
  else
    Editor->ScrollBars = ssBoth;            // unwrapped can need both
  WordWrap1->Checked = Editor->WordWrap;    // check menu item to match property
}
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

Adding the Clipboard Object

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. *TClipboard* object encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.

To add the Clipboard object to an application:

1. Select the unit that will use the clipboard.
2. Search for the **implementation** reserved word.
3. Add *Clipbrd* to the **uses** clause below **implementation**.

- If there is already a **uses** clause in the **implementation** part, add *Clipbrd* to the end of it.
- If there is not already a **uses** clause, add one that says

```
[Delphi]
    uses Clipbrd;
```

```
[C++]
#include <vcl\Clipbrd.hpp>
```

For example, in an application with a child window, the uses clause in the unit's implementation part might look like this:

```
[Delphi]
uses
  MDIFrame, Clipbrd;
```

Selecting Text

For text in an edit control, before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

The table below lists properties commonly used to handle selected text.

Properties of selected text

Property	Description
SelText	Contains a string representing the selected text in the component.
SelLength	Contains the length of a selected string.
SelStart	Contains the starting position of a string relative to the beginning of an edit control's text buffer.

For example, the following *OnFind* event handler searches a Memo component for the text specified in the *FindText* property of a find dialog component. If found, the first occurrence of the text in Memo1 is selected.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
I, J, PosReturn, SkipChars: Integer;
begin
for I := 0 to Memo1.Lines.Count do
begin
PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
if PosReturn <> 0 then {found!}
begin
Skipchars := 0;
for J := 0 to I - 1 do
Skipchars := Skipchars + Length(Memo1.Lines[J]);
SkipChars := SkipChars + (I*2);
SkipChars := SkipChars + PosReturn - 1;
      Memo1.SetFocus;
Memo1.SelStart := SkipChars;
Memo1.SelLength := Length(FindDialog1.FindText);
Break;
end;
end;
end;
```

Selecting All Text

The *SelectAll* method selects the entire contents of an edit control, such as a rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example:

```
[Delphi]
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll;  { select all text in RichEdit }
end;
```

```
[C++]
void __fastcall TMainForm::SelectAll(TObject *Sender)
{
  RichEdit1->SelectAll();     // select all text in RichEdit
}
```

Cutting, Copying, and Pasting Text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard.

To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the **Edit** ▸ **Cut**, **Edit** ▸ **Copy**, and **Edit** ▸ **Paste** commands, respectively:

```
[Delphi]
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

```
[C++]
void __fastcall TMainForm::EditCutClick(TObject* Sender)
{    RichEdit1->CutToClipboard();
}
void __fastcall TMainForm::EditCopyClick(TObject* Sender)
{    RichEdit1->CopyToClipboard();
}
void __fastcall TMainForm::EditPasteClick(TObject* Sender)
{    RichEdit1->PasteFromClipboard();
}
```

Deleting Selected Text

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
[Delphi]
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

```
[C++]
void __fastcall TMainForm::EditDeleteClick(TObject *Sender)
{
  RichEdit1->ClearSelection();
}
```

Disabling Menu Items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *RichEdit1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```
[Delphi]
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);  {enable or disable the
Paste                                                menu item}
  HasSelection := Editor.SelLength > 0;  { True if text is selected }
  Cut1.Enabled := HasSelection;  { enable menu items if HasSelection is True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
```

```
[C++]
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;  // enable menu items if HasSelection is true
  Copy1->Enabled = HasSelection;
  Delete1->Enabled = HasSelection;
}
```

The *HasFormat* method (*Provides* method in CLX applications) of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. By calling *HasFormat* with the parameter *CF_TEXT*, you can determine whether the clipboard contains any text, and enable or disable the Paste item as appropriate.

Note: In CLX applications, use the *Provides* method. In this case, the text is generic. You can specify the type of text using a subtype such as text/plain for plain text or text/html for html.

Providing a Pop-up Menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form:

1. Place a pop-up menu component on the form.
2. Use the Menu Designer to define the items for the pop-up menu.
3. Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
4. Attach handlers to the *OnClick* events of the pop-up menu items.

Handling the OnPopup Event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them:

1. Select the pop-up menu component.
2. Attach an event handler to its *OnPopup* event.
3. Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in Disabling menu items is attached to the pop-up menu component's OnPopup event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

```
[Delphi]
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Add this line}
  Delete1.Enabled := HasSelection;
end;
```

```
[C++]
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  Paste2->Enabled = Paste1->Enabled;            // add this line
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;    // enable menu items if HasSelection is true
  Cut2->Enabled = HasSelection;    // add this line
  Copy1->Enabled = HasSelection;
  Copy2->Enabled = HasSelection;   // add this line
```

```
  Delete1->Enabled = HasSelection;
}
```

Adding Graphics to Controls

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and toolbars. Instead of using the standard method of drawing a control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see Adding images to menu items..

All owner-draw controls contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in the list to make it easy to use that object when drawing items.

To create an owner-draw control:

1. Indicating that a control is owner-drawn.
2. Adding graphical objects to a string list.
3. Drawing owner-drawn items.

Indicating That a Control Is Owner-drawn

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and toolbars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as *OnCustomDraw* or *OnAdvancedCustomDraw*.

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. List views and tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called fixed and *variable*, as the following table describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the ItemHeight property.	*lbOwnerDrawFixed*,*csOwnerDrawFixed*
Variable	Each item might have a different height, determined by the data at runtime.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

Adding Graphical Objects to a String List

Every string list has the ability to hold a list of objects in addition to its list of strings. You can also add graphical objects of varying sizes to a string list.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

Note that you can also organize graphical objects using an image list by creating a *TImageList*. However, these images must all be the same size. See Adding images to menu items for an example of setting up an image list.

Adding Images to an Application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example,

To store bitmaps for owner-draw controls in hidden image controls:

1. Add image controls to the main form.
2. Set their *Name* properties.
3. Set the Visible property for each image control to *False*.
4. Set the *Picture* property of each image to the desired bitmap using the Picture editor from the **Object Inspector**.

The image controls are invisible when you run the application. The image is stored with the form so it doesn't have to be loaded from a file at runtime.

Adding Images to a String List

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```
[Delphi]
procedure TFMForm.FormCreate(Sender: TObject);
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do  { iterate through all possible drives }
  begin
    case GetDriveType(Drive + ':/') of  { positive values mean valid drives }
     DRIVE_REMOVABLE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
     DRIVE_FIXED:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
     DRIVE_REMOTE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { current drive? }
     DriveTabSet.TabIndex := AddedIndex;  { then make that current tab }
  end;
end;
```

```
[C++]
void __fastcall TFMForm::FormCreate(TObject *Sender)
{
  int AddedIndex;
  char DriveName[4] = "A:\\";
  for (char Drive = "A"; Drive <= "Z"; Drive++) // try all possible drives
  {
    DriveName[0] = Drive;
    switch (GetDriveType(DriveName))
    {
      case DRIVE_REMOVABLE:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Floppy->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
      case DRIVE_FIXED:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Fixed->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
      case DRIVE_REMOTE:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Network->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
    }
    if ((reinterpret_cast<int>(Drive - "A")) == getdisk()) // current drive?
      DriveList->ItemIndex = AddedIndex; // then make that the current list item
  }
}
```

Drawing Owner-drawn Items

When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

1. Size the item, if needed.

   Items of the same size (for example, with a list box style of *IsOwnerDrawFixed*), do not require sizing.

2. Draw the item.

Sizing Owner-draw Items

Before giving your application the chance to draw each item in a variable owner-draw control, the control receives a measure-item event, which is of type *TMeasureItemEvent*. *TMeasureItemEvent* tells the application where the item appears on the control.

Delphi determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be large enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the height of that item. The height is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```
[Delphi]
procedure TFMForm.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);  { note that Height is a var parameter}
var
  BitmapHeight: Integer;
begin
  BitmapHeight := TBitmap(ListBox1.Items.Objects[Index]).Height;
  { make sure the item height has enough room, plus two }
  Height := Max(Height, Bitmap Height +2);
end;
```

```
[C++]
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control, int Index,
  int &Height)      // note that Height is passed by reference
{
  int BitmapHeight = (dynamic_cast<TBitmap *>(ListBox1->Items->Objects[Index]))->Height + 2;
  // make sure list item has enough room for bitmap (plus two)
  Height = Max(Height, Bitmap Height +2);
}
```

Note: You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

Drawing Owner-draw Items

When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a part of the item.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*

- *OnCustomDraw*, such as *OnCustomDrawItem*
- *OnAdvancedCustomDraw*, such as *OnAdvancedCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```
[Delphi]
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
      Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
      TextOut(R.Left + 2 + Bitmap.Width,  { position text }
        R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the bitmap }
  end;
end;
```

```
[C++]
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)
  TBitmap *Bitmap = (TBitmap *)ListBox1->Items->Objects[Index];
  ListBox1->Canvas->Draw(R.Left, R.Top + 2, Bitmap); // draw the bitmap
  ListBox1->Canvas->TextOut(R.Left + Bitmap->Width + 2, R.Top + 2,
    ListBox1->Items->Strings[Index]);        // and write the text to its right
}
```

Building applications, components, and libraries

Creating Applications

The most common types of applications you can design and build are:

- GUI applications
- Console applications
- Service applications
- Packages and DLLs

GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code.

You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to Creating packages and DLLs.

GUI Applications

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable.

The IDE offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

User Interface Models

Any form can be implemented as a single document interface (SDI) or multiple document interface (MDI) form. An SDI application normally contains a single document view. In an MDI application, more than one document or child

window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors.

For more information on developing the UI for an application, see Developing the application user interface.

SDI Applications

To create a new SDI application:

1. Choose **File** ▸ **New** ▸ **Other** to bring up the New Items dialog.
2. Click on the Projects page and double-click SDI Application.
3. Click OK.

By default, the *FormStyle* property of your *Form* object is set to fsNormal, so that the IDE assumes that all new applications are SDI applications.

MDI Applications

To create a new MDI application using a wizard:

1. Choose **File** ▸ **New** ▸ **Other** to bring up the New Items dialog.
2. Click on the Projects page and double-click MDI Application.
3. Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIChild*) or main form (*fsMDIForm*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

MDI applications often include a Window pop-up on the main menu that has items such as Cascade and Tile for viewing multiple windows in various styles. When a child window is minimized, its icon is located in the MDI parent form.

To create a new MDI application without using a wizard:

1. Create the main window form or MDI parent window. Set its *FormStyle* property to *fsMDIForm*.
2. Create a menu for the main window that includes **File** ▸ **Open**, **File** ▸ **Save**, and Window which has Cascade, Tile, and Arrange All items.
3. Create the MDI child forms and set their *FormStyle* properties to *fsMDIChild*.

Setting IDE, Project, and Compiler Options

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE. To specify various options for your project, choose **Project** ▸ **Options**.

Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom left of the window. All new projects will use the current options selected by default.

Code Templates

Code templates are commonly used *skeleton* structures that you can add to your source code and then fill in. You can also use standard code templates such as those for array, class, and function declarations, and many statements.

You can also write your own templates for coding structures that you often use. For example, if you want to use a **for** loop in your code, you could insert the following template:

```
[Delphi]
for :=  to  do
begin
end;
```

```
[C++]
for (; ;)
{
}
```

To insert a code template in the Code editor, press *Ctrl-j* and select the template you want to use. You can also add your own templates to this collection.

To add a template:

1. Choose **Tools** ▸ **Options** ▸ **Editor Options**.
2. Click the Source Options tab and then the Edit Code Templates button.
3. In the Templates section, click Add.
4. Type a name for the template after Shortcut name, enter a brief description of the new template, and click OK.
5. Add the template code to the Code text box.
6. Click OK.

Console Applications

Console applications are 32-bit programs that run without a graphical interface, in a console window. These applications typically don't require much user input and perform a limited set of functions. Any application that contains:

```
{$APPTYPE CONSOLE}
```

in the code opens a console window of its own.

To create a new console application, choose **File** ▸ **New** ▸ **Other**. Select Delphi Projects and double-click Console Application from the New Items dialog box.

The IDE then creates a project file for this type of source file and displays the Code editor.

Console applications should make sure that no exceptions escape from the program scope. Otherwise, when the program terminates, the Windows operating system displays a modal dialog with exception information. For example, your application should include exception handling such as shown in the following code:

```
program ConsoleExceptionHandling;
{$APPTYPE CONSOLE}
uses
SysUtils;
procedure ExecuteProgram;
begin
  //Program does something
  raise Exception.Create('Unforeseen exception');
end;
begin
  try
    ExecuteProgram;
  except
//Handle error condition
    WriteIn('Program terminated due to an exception');
    //Set ExitCode <> 0 to flag error condition (by convention)
    ExitCode := 1;
  end;
end.
```

Users can terminate console applications in one of the following ways:

- Click the Close (X) button.
- Press Ctrl+C.
- Press Ctrl+Break.
- Log off.

Depending on which way the user chooses, the application is terminated forcefully, the process is not shut down cleanly, and the finalization section isn't run. Use the Windows API *SetConsoleCtrlHandler* function for options for handling these user termination requests.

Service Applications

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A Web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service:

1. Choose **File** ▸ **New** ▸ **Other**, and double-click Service Application in the New Items dialog box. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.
2. A Service window appears that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the **Object Inspector**.
3. You can add additional services to your service application by choosing **File** ▸ **New** ▸ **Other**, and double-click Service in the New Items dialog box. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.

4 Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the /INSTALL option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the /SILENT option.

To uninstall the services, run it from the command line using the /UNINSTALL option. (You can also use the /SILENT option to suppress the confirmation message when uninstalling).

Note: This service has a *TServerSocket* whose port is set to 80. This is the default port for Web browsers to make requests to Web servers and for Web servers to make responses to Web browsers. This particular example produces a text document in the C:\Temp directory called WebLog*xxx*.log (where *xxx* is the ThreadID). There should be only one server listening on any given port, so if you have a Web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a Web browser on the local machine and for the address, type 'localhost' (with no quotes). The browser will time out eventually, but you should now have a file called Weblog*xxx*.log in the C:\Temp directory.

To create the example:

1 Choose **File** ▸ **New** ▸ **Other** and select Service Application from the New Items dialog box. The Service1 window appears.

2 From the Internet category of the **Tool palette**, add a ServerSocket component to the service window (Service1).

3 Add a private data member of type TMemoryStream to the TService1 class. The interface section of your unit should now look like this:

```
[Delphi]

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;
type
    TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;
var
  Service1: TService1;
```

```
[C++]
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
```

```
#include <SvcMgr.hpp>
#include <ScktComp.hpp>
//---------------------------------------------------------------------------
class TService1 : public TService
{
__published:
TServerSocket *ServerSocket1;
private:
TMemoryStream *Stream; // add this line here
public:
__fastcall TService1(TComponent* Owner);
PServiceController __fastcall GetServiceController(void);
friend void __stdcall ServiceController(unsigned CtrlCode);
};
//---------------------------------------------------------------------------
extern PACKAGE TService1 *Service1;
//---------------------------------------------------------------------------
#endif
```

4 Select ServerSocket1, the component you added in step 1. In the **Object Inspector**, double-click the OnClientRead event and add the following event handler:

```
[Delphi]

procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
begin
  Buffer := nil;
while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
                  FreeMem(Buffer);
    end;
  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
```

```
[C++]
void __fastcall TService1::ServerSocket1ClientRead(TObject *Sender,
TCustomWinSocket *Socket)
{
char *Buffer = NULL;
int len = Socket->ReceiveLength();
while (len > 0)
{
try
{
Buffer = (char *)malloc(len);
Socket->ReceiveBuf((void *)Buffer, len);
Stream->Write(Buffer, len);
}
__finally
{
```

```
free(Buffer);
}
Stream->Seek(0, soFromBeginning);
AnsiString LogFile = "C:\\Temp\\WebLog";
LogFile = LogFile + IntToStr(ServiceThread->ThreadID) + ".log";
Stream->SaveToFile(LogFile);
}
}
```

5 Finally, select Service1 by clicking in the window's client area (but not on the ServiceSocket). In the **Object Inspector**, double click the OnExecute event and add the following event handler:

```
[Delphi]

procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
  try
    ServerSocket1.Port := 80; // WWW port
    ServerSocket1.Active := True;
    while not Terminated do begin
      ServiceThread.ProcessRequests(True);
    end;
    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;
```

```
[C++]
void __fastcall TService1::Service1Execute(TService *Sender)
{
Stream = new TMemoryStream();
try
{
ServerSocket1->Port = 80; // WWW port
ServerSocket1->Active = true;
while (!Terminated)
ServiceThread->ProcessRequests(true);
ServerSocket1->Active = false;
}
__finally
{
delete Stream;
}
}
```

When writing your service application, you should be aware of:

- Service threads
- Service name properties
- Debugging service applications

Note: Service applications are not available for cross-platform applications.

Service Threads

Each service has its own thread (TServiceThread), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TService OnExecute* event handler. The service thread has its own Execute method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests.

Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from TThread, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

Note: This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

To create the example:

1. Choose **File** ▸ **New** ▸ **Other** and double-click Service Application in the New Items dialog. The Service1 window appears.
2. In the interface section of your unit, declare a new descendant of TThread named TSparkyThread. This is the thread that does the work for your service. The declaration should appear as follows:

```
[Delphi]

TSparkyThread = class(TThread)
  public
    procedure Execute; override;
end;
```

```
[C++]
class TSparkyThread : public TThread
{
private:
protected:
  void __fastcall Execute();
public:
__fastcall TSparkyThread(bool CreateSuspended);
};
```

3. In the implementation section of your unit, create a global variable for a TSparkyThread instance:

```
[Delphi]

var
  SparkyThread: TSparkyThread;
```

```
[C++]
TSparkyThread *SparkyThread;// Add this code as the constructor
__fastcall TSparkyThread::TSparkyThread(bool CreateSuspended)
: TThread(CreateSuspended)
{
}
```

4 In the implementation section for the TSparkyThread Execute method (the thread function), add the following code:

```
[Delphi]

procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

```
[C++]
void __fastcall TSparkyThread::Execute()
{
  while (!Terminated)
  {
    Beep();
    Sleep(500);
  }
}
```

5 Select the Service window (Service1), and double-click the OnStart event in the **Object Inspector**. Add the following OnStart event handler:

```
[Delphi]

procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

```
[C++]
void __fastcall TService1::Service1Start(TService *Sender, bool &Started)
{
  SparkyThread = new TSparkyThread(false);
  Started = true;
}
```

6 Double-click the OnContinue event in the **Object Inspector**. Add the following OnContinue event handler:

```
[Delphi]

procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

```
[C++]
void __fastcall TService1::Service1Continue(TService *Sender, bool &Continued)
{
```

```
  SparkyThread->Resume();
  Continued = true;
}
```

7 Double-click the OnPause event in the **Object Inspector**. Add the following OnPause event handler:

```
[Delphi]

procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

```
[C++]
void __fastcall TService1::Service1Pause(TService *Sender, bool &Paused)
{
  SparkyThread->Suspend();
  Paused = true;
}
```

8 Finally, double-click the OnStop event in the **Object Inspector** and add the following OnStop event handler:

```
[Delphi]

procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

```
[C++]
void __fastcall TService1::Service1Stop(TService *Sender, bool &Stopped)
{
  SparkyThread->Terminate();
  Stopped = true;
}
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

Service Name Properties

The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This topic describes the differences.

Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties.

TDependency properties

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

TService name properties

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

TService's DisplayName is the name displayed in the Service Manager window. This often differs from the actual service name (*TService.Name, TDependency.DisplayName, TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

Debugging Service Applications

You can debug service applications by attaching to the service application process when it is already running (that is, by starting the service first, and then attaching to the debugger). To attach to the service application process, choose **Run** ▸ **Attach To Process**, and select the service application in the resulting dialog.

In some cases, this approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

To debug:

1. First create a key called **Image File Execution Options** in the following registry location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
```

2. Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG_SZ, named Debugger. Use the full path to bds.exe as the string value.
3. In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

On Windows NT systems, you can use another approach for debugging service applications. However, this approach can be tricky, because it requires short time intervals:

For Windows NT:

1. First, launch the application in the debugger. Wait a few seconds until it has finished loading.
2. Quickly start the service from the Control Panel or from the command line:

```
start MyServ
```

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

Creating Packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects.

DLLs and libraries should handle all exceptions to prevent the display of errors and warnings through Windows dialogs.

The following compiler directives can be placed in library project files:

Compiler directives for libraries

Compiler Directive	Description
{$LIBPREFIX 'string'}	Adds a specified prefix to the output file name. For example, you could specify {$LIBPREFIX 'dcl'} for a design-time package, or use {$LIBPREFIX''} to eliminate the prefix entirely.
{$LIBSUFFIX 'string'}	Adds a specified suffix to the output file name before the extension. For example, use {$LIBSUFFIX '-2.1.3'} in something.pas to generate something-2.1.3.bpl.
{$LIBVERSION 'string'}	Adds a second extension to the output file name after the .bpl extension. For example, use {$LIBVERSION '2.1.3'} in something.pas to generate something.bpl.2.1.3.

Packages are special DLLs used by Delphi applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

For more information on packages, see Working with packages and components.

When to Use Packages and DLLs

For most applications, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a Web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

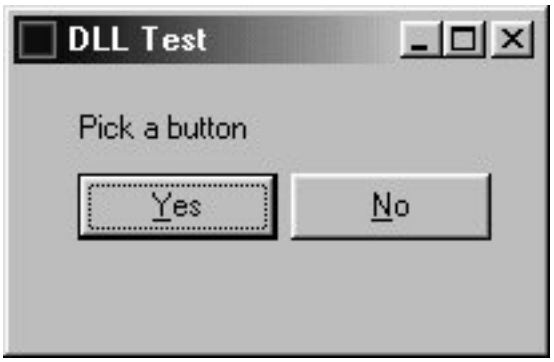
However, if your application includes VisualCLX, you must use packages instead of DLLs. Only packages can manage the startup and shut down of the Qt shared libraries.

You cannot pass Delphi runtime type information (RTTI) across DLLs or from a DLL to an executable. If you pass an object from one DLL to another DLL or an executable, you will not be able to use the **is** or **as** operators with the passed object. This is because the **is** and **as** operators need to compare RTTI. If you need to pass objects from a library, use packages instead, as these can share RTTI. Similarly, you should use packages instead of DLLs in Web Services because they are rely on Delphi RTTI.

Creating DLLs Containing VCL Components (C++)

One of the strengths of DLLs is that a DLL created with one development tool can often be used by application written using a different development tool. When your DLL contains VCL or CLX components (such as forms) that are to be used by the calling application, you need to provide exported interface routines that use standard calling conventions, avoid C++ name mangling, and do not require the calling application to support the VCL and CLX libraries in order to work. To create VCL or CLX components that can be exported, use runtime packages.

For example, suppose you want to create a DLL to display the following simple dialog box:



The code for the dialog box DLL is as follows:

```
// DLLMAIN.H
//---------------------------------------------------------------------------
#ifndef dllMainH
#define dllMainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TYesNoDialog : public TForm
{
    __published:    // IDE-managed Components
        TLabel *LabelText;
        TButton *YesButton;
        TButton *NoButton;
        void __fastcall YesButtonClick(TObject *Sender);
        void __fastcall NoButtonClick(TObject *Sender);
    private:        // User declarations
        bool returnValue;
    public:         // User declarations
        virtual __fastcall TYesNoDialog(TComponent *Owner);
        bool __fastcall GetReturnValue();
};
// exported interface function
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();
//---------------------------------------------------------------------------
extern TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
#endif
// DLLMAIN.CPP
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "dllMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
__fastcall TYesNoDialog::TYesNoDialog(TComponent *Owner)
    : TForm(Owner)
{
    returnValue = false;
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
```

```
{
    returnValue = true;
    Close();
}
//---------------------------------------------------------------------
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
{
    returnValue = false;
    Close();
}
//---------------------------------------------------------------------
bool __fastcall TYesNoDialog::GetReturnValue()
{
    return returnValue;
}
//---------------------------------------------------------------------
// exported standard C++ interface function that calls into VCL
bool InvokeYesNoDialog()
{
    bool returnValue;
    TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);
    YesNoDialog->ShowModal();
    returnValue = YesNoDialog->GetReturnValue();
    delete YesNoDialog;
    return returnValue;
}
//---------------------------------------------------------------------
```

The code in this example displays the dialog and stores the value true in the private data member `returnValue` if the "Yes" button is pressed. Otherwise, `returnValue` is false. The public `GetReturnValue()` function retrieves the current value of `returnValue`.

To invoke the dialog and determine which button was pressed, the calling application calls the exported function `InvokeYesNoDialog()`. This function is declared in DLLMAIN.H as an exported function using C linkage (to avoid C++ name mangling) and the standard C calling convention. The function is defined in DLLMAIN.CPP.

By using a standard C function as the interface into the DLL, any calling application, whether or not it was created with Developer Studio 2006, can use the DLL. The VCL and CLX functionality required to support the dialog is linked into the DLL itself, and the calling application does not need to know anything about it.

Note: When creating a DLL that uses the VCL or CLX, the required VCL or CLX components are linked into the DLL resulting in a certain amount of overhead. The impact of this overhead on the overall size of the application can be minimized by combining several components into one DLL that only needs one copy of the VCL and CLX support components.

Using DLLs in Developer Studio 2006 (C++)

A Windows DLL can be used in a Developer Studio 2006 application just as it would be in any C++ application.

To statically load a DLL when your C++ application is loaded, link the import library file for that DLL into your C++ application at link time. To add an import library to a C++ application, choose **Project** ▸ **Add to Project** and select the .lib file you want to add to the libraries to be linked.

The exported functions of that DLL then become available for use by your application. Prototype the DLL functions your application uses with the __declspec (dllimport) modifier:

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

To dynamically load a DLL during the run of a C++ application, include the import library, just as you would for static loading, and set the **Delay load a DLL** option for ilink32. You can also use the Windows API function `LoadLibrary()` to load the DLL, then use the API function `GetProcAddress()` to obtain pointers to the individual functions you want to use.

Linking DLLs

You can set the linker options for your DLL on the **Linker** page of the **Project Options** dialog box. The default check box on this page also creates an import library for your DLL. If compiling from the command line, invoke the linker, ilink32.exe, with the -Tpd switch. For example:

```
ilink32 /c /aa /Tpd c0d32.obj mydll.obj, mydll.dll, mydll.map, import32.lib cw32mt.lib
```

If you need an import library, use the -Gi switch to generate an import library.

You can optionally create an import library with the command line utility implib.exe. For more information on implib.exe, type implib —h at the command line.

Writing Database Applications

You can create advanced database applications using tools to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, PostgreSQL, and DB2 while providing transparent data sharing between applications.

The **Tool palette** includes many components for accessing databases and representing the information they contain. The database components are grouped according to the data access mechanism and function.

Database pages on the Tool palette

Palette page	Contents
BDE	Components that use the Borland Database Engine (BDE), a large API for interacting with databases. The BDE supports the broadest range of functions and comes with the most supporting utilities including Database Desktop and Database Explorer . See Using the Borland Database Engine for details.
ADO	Components that use ActiveX Data Objects (ADO), developed by Microsoft, to access database information. Many ADO drivers are available for connecting to different database servers. ADO-based components let you integrate your application into an ADO-based environment. See Working with ADO Components for details.
dbExpress	Cross-platform components that use dbExpress to access database information. dbExpress drivers provide fast access to databases but need to be used with *TClientDataSet* and *TDataSetProvider* to perform updates. See Using Unidirectional Datasets for details.
InterBase	Components that access InterBase databases directly, without going through a separate engine layer.
Data Access	Components that can be used with any data access mechanism such as *TClientDataSet* and *TDataSetProvider*. See Using Client Datasets: Overview for information about client datasets. See Using Provider Components for information about providers.
Data Controls	Data-aware controls that can access information from a data source. See Using Data Controls for details.

When designing a database application, you must decide which data access mechanism to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

Refer to Designing database applications for details on what type of database support is available and considerations when designing database client applications and application servers.

Note: Not all editions of Delphi include database support.

Distributing Database Applications

You can create distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and SOAP.

For more information about building distributed database applications, see Creating Multi-tiered Applications.

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see Deploying Database Applications.

Creating Web Server Applications

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following technologies:

- Web Broker
- WebSnap
- IntraWeb
- Web Services

Creating Web Broker Applications

You can use Web Broker (also called NetCLX architecture) to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet category of the **Tool palette** enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

To create a new Web server application using the Web Broker architecture, choose **File** ▸ **New** ▸ **Other**. In the New Items dialog box, select the Delphi Projects tab. Then select the New tab and double-click the Web Server Application. Then select the Web server application type:

Web server applications

Web server application type	Description
ISAPI and NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client. Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.
Apache Shared Module (DLL)	Selecting this type of application sets up your project as a DLL. Apache Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.
Web App Debugger stand-alone executable	Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are

	executable files loaded by the Web server. This type of application is not intended for deployment.

CGI applications use more system resources on the server, so complex applications are better created as ISAPI, NSAPI, or Apache DLL applications. When writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (DLL) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see Creating Internet Server Applications.

Creating WebSnap Applications

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other MIME content for Web pages. WebSnap is for server-side development.

To create a new WebSnap application, select **File** ▸ **New** ▸ **Other** and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Apache). See the table in the topic Using Web Broker for details.

If you want to do client-side scripting instead of server-side scripting, you can use the InternetExpress technology. For more information on InternetExpress, see Building Web applications using InternetExpress.

For more information on WebSnap, see Creating Internet Server Applications.

Creating Web Services Applications

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and servers to publish Web Services for use by arbitrary clients.

Refer to Using Web Services for more information on Web Services.

Writing Applications Using COM

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Using COM and DCOM

Various classes and wizards that make it easy to create COM, OLE, or ActiveX applications. You can create COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX

controls, or ActiveForms. COM also severs as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories.

Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win9x Shell extensions and DirectX multimedia support. Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

For more information on COM and Active X controls, see Overview of COM technologies,Creating an ActiveX Control and Distributing a Client Application as an ActiveX Control.

For more information on DCOM, see Using DCOM connections.

Using MTS and COM+

COM applications can be augmented with special services for managing objects in a large distributed environment. These services include transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) on versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later).

For more information on MTS and COM+, see Creating MTS or COM+ objects and Using transactional data modules.

Using Data Modules

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose.

- Standard data modules are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see Creating and Editing Data Modules.
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see Adding a remote data module to an application server project.
- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Creating Internet Server Applications for more information about using Web modules.
- Applet modules form the basis of control panel applets. In addition to holding the nonvisual controls that implement the control panel applet, they define the properties that determine how the applet's icon appears in the control panel and include the events that are called when users execute the applet.
- Services encapsulate individual services in an NT service application. In addition to holding any nonvisual controls used to implement a service, services include the events that are called when the service is started or stopped. For more information about services, see Service Applications.

Creating and Editing Standard Data Modules

To create a standard data module for a project, choose **File** ▸ **New** ▸ **Data Module**. The IDE opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time, a data module looks like a standard form with a white background and no alignment grid. As with forms, you can place nonvisual components from the **Tool palette** onto a module, and edit their properties in the **Object Inspector**. You can resize a data module to accommodate the components you add to it.

You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

Context menu options for data modules

Menu item	Purpose
Edit	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
Position	Aligns nonvisual components to the module's invisible grid (*Align To Grid*) *or* according to criteria you supply in the Alignment dialog box (*Align*).
Tab Order	Enables you to change the order that the focus jumps from component to component when you press the tab key.
Creation Order	Enables you to change the order that data access components are created at start-up.
Revert to Inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.
Text DFM	Toggles between the formats (binary or text) in which this particular form file is saved.

Naming a Data Module and Its Unit File

The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2. " The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

1. Select the module.
2. Edit the Name property for the module in the **Object Inspector**.

The new name for the module appears in the title bar when the *Name* property in the **Object Inspector** no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module, select the unit file.

Placing and Naming Components

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate category of the **Tool palette**, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, the module assigns it a generic name that identifies what kind of component it is, followed by a 1. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

1. Select the component.
2. Edit the component's *Name* property in the **Object Inspector**.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the **Object Inspector** no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource*) and a table component (*TClientDataSet*). When you place these components in your data module, the module assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTable*.

Using Component Properties and Events in a Data Module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a TDataSource component has three possible events: *OnDataChange*, *OnStateChange*, and *OnUpdateData*. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

Creating Business Rules in a Data Module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module.

The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
```

```
    Orders: TClientDataSet;
    .
    .
    .
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;
var
  CustomerData: TCustomerData;
```

The procedures and functions you write should follow in the implementation section of the code for the module.

Accessing a Data Module from a Form

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.
- Click the form's unit file, choose **File** ▸ **Use Unit**, and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the Fields editor and drag any existing fields from the editor onto the form. The IDE prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, Delphi adds a new data source component, *DataSource1*, to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1*, in the data module.

You can define the data source *before* you drag a field to the form by adding a TDataSource component to the data module. Set the data source's *DataSet* property to *ClientDataSet1*. After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

Adding a Remote Data Module to an Application Server Project

Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

1. Choose **File** ▸ **New** ▸ **Other**.
2. Select the ActiveX page in the New Items dialog box.
3. Double-click the Remote Data Module icon to open the Remote Data Module wizard.

Once you add a remote data module to a project, use it just like a standard data module.

For more information about multi-tiered database applications, see Creating multi-tiered applications.

Using the Object Repository

The Object Repository (**Tools** ▸ **Options** ▸ **Repository (under Translation Tools Options)**) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The Object Repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

Sharing Items Within a Project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (**File** ▸ **New** ▸ **Other**), you"ll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

Adding Items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository.

To add an item to the Object Repository

1. If the item is a project or is in a project, open the project.
2. For a project, choose **Project** ▸ **Add To Repository**. For a form or data module, right-click the item and choose Add To Repository.
3. Type a description, title, and author.
4. Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, the Object Repository creates a new page.
5. Choose Browse to select an icon to represent the object in the Object Repository.
6. Choose OK.

Sharing Objects in a Team Environment

You can share objects with your workgroup or development team by making a repository available over a network.

To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1. Choose **Tools** ▸ **Options** ▸ **Environment Options**.
2. On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the Repository, a DELPHI32.DRO file is created in the Shared Repository directory if one doesn't exist already.

Using an Object Repository Item in a Project

To access items in the Object Repository, choose **File** ▸ **New** ▸ **Other**. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy
- Inherit
- Use

Copying an Item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

Inheriting an Item

Choose Inherit to derive a new class from the selected item and add the new class to your project. When you recompile your project, any changes that have been made to the item Will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

Using an Item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

Using Project Templates

Templates are predesigned projects that you can use as starting points for your own work.

To create a new project from a template:

1. Choose **File** ▸ **New** ▸ **Other** to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog, specify a directory for the new project's files.

The template files are copied to the specified directory, where you can modify them. The original project template is unaffected by your changes.

To add projects and project templates to the Object Repository, see Adding items to the Object Repository.

Modifying Shared Items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

Enabling Help in Applications

VCL applications support displaying Help using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

VCL applications provide an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support.

If more than one viewer supports the keyword, as would be the case in an application that had registered viewers for both WinHelp and HyperHelp on Windows, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered.

Help System Interfaces

The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in the HelpIntfs.pas, which also contains the implementation of the Help Manager.

ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, "IntToStr" might be a keyword in the Help system, but "String manipulation routines" could be the name of a topic).

ISpecialWinHelpViewer provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

IHelpManager provides a mechanism for the Help viewer to communicate back to the application's Help Manager and request additional information. *IHelpManager* is obtained at the time the Help viewer registers itself.

IHelpSystem provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

IHelpSelector provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display

a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

Implementing ICustomHelpViewer

The ICustomHelpViewer interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

For information on ICustomHelpViewer methods, see

- Communicating with the Help Manager
- Displaying keyword-based Help
- Asking the Help Manager for information

Communicating with the Help Manager

The ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- SoftShutDown

The Help Manager calls through these functions in the following circumstances:

- *ICustomHelpViewer.GetViewerName* : *String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)
- *ICustomHelpViewer.ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.
- *ICustomHelpViewer.SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the Help system (for example, windows displaying Help information) without unloading the viewer.

Asking the Help Manager for Information

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things:

- A request for the window handle of the currently active control.
- A request for the name of the Help file which the Help Manager believes should contain help for the currently active control.
- A request for the path to that Help file.
- A notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

IHelpManager.GetHandle : *LongInt* is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

IHelpManager.GetHelpFile: *String* is called by the Help viewer if it needs to know the name of the Help file which the currently active control believes contains its Help.

IHelpManager.Release is called to notify the Help Manager when a Help viewer is disconnecting. It should never be called in response to a request through *ICustomHelpViewer.ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

Displaying Keyword-based Help

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier.

Note: Numeric Help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system; while CLX supports them, they are not recommended for use in CLX applications because most Linux Help systems do not understand them.

ICustomHelpViewer implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

ICustomHelpViewer provides three methods for handling keyword-based Help:

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
[Delphi]
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
```

```
[C++]
int__fastcall ICustomHelpViewer::UnderstandsKeyword(const AnsiString HelpString)
```

is the first of the three methods called by the Help Manager, which will call each registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this—inside the IDE, the HyperHelp viewer maintains its own index and searches it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
[Delphi]
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

```
[C++]
Classes::TStringList*__fastcall ICustomHelpViewer::GetHelpStrings(const AnsiString
HelpString)
```

is called by the Help Manager if more than one viewer can provide Help on a topic. The viewer is expected to return a *TStringList*, which is freed by the Help Manager. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the WinHelp viewer on Windows and the HyperHelp viewer on Linux, the string list always contains exactly one entry. HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication. In the case of the Man page viewer (Linux), the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
[Delphi]
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

```
[C++]
void__fastcall ICustomHelpViewer::ShowHelp(const AnsiString HelpString)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless the *UnderstandsKeyword* method is invoked first.

Displaying Tables of Contents

ICustomHelpViewer provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*.

It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to WinHelp on Windows and HyperHelp on Linux. It is not reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through *CanShowTableOfContents* with the answer *True*, and then ignore requests through *ShowTableOfContents*.

Implementing IExtendedHelpViewer

ICustomHelpViewer only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications can talk to systems requiring context-based Help by extending the object that implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches. The built-in WinHelp viewer does this for you automatically.

IExtendedHelpViewer exposes four functions. Two of them—*UnderstandsContext* and *DisplayHelpByContext*—are used to support context-based Help; the other two—*UnderstandsTopic* and *DisplayTopic*—are used to support topics.

When an application user presses F1, the Help Manager calls

```
[Delphi]
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

```
[C++]
int__fastcall IExtendedHelpViewer::UnderstandsContext(const int ContextID, AnsiString
HelpFileName)
```

and the currently activated control supports context-based, rather than keyword-based Help. As with *ICustomHelpViewer.UnderstandsKeyword*, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *ICustomHelpViewer.UnderstandsKeyword*, however, if more than one viewer supports a specified context, the first registered viewer with support for a given context is invoked.

The Help Manager calls

```
[Delphi]
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
```

```
[C++]
void__fastcall IExtendedHelpViewer::DisplayHelpByContext(const int ContextID, AnsiString
HelpFileName)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
[Delphi]
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

```
[C++]
bool__fastcall IExtendedHelpViewer::UnderstandsTopic(const AnsiString Topic)
```

is used to poll the Help viewers asking if they support a topic;

```
[Delphi]
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

```
[C++]
void__fastcall IExtendedHelpViewer::DisplayTopic(const AnsiString Topic)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

Implementing IHelpSelector

IHelpSelector is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims

to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a TStrings containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStringList*) that represents the selected string; the *TStringList* is then freed by the Help Manager.

Note: The Help Manager may get confused if the strings are rearranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

Registering Help System Objects

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer,* and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to:

- Register the Help viewer.
- Register the Help Selector.

Registering Help viewers

The unit that contains the object implementation must use HelpIntfs. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by the HelpIntfs unit, which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

```
[Delphi]
```

Registering Help selectors

The unit that contains the object implementation must use either Forms in the VCL or QForms in CLX. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must register the Help selector through the *HelpSystem* property of the global Application object:

```
[Delphi]
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

```
[C++]
Application->HelpSystem->AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

Using Help in a VCL Application

The following sections explain how to use Help within a VCL application.

- How TApplication Processes VCL Help
- How VCL controls process Help
- Calling a Help system directly
- Using IHelpSystem

How TApplication Processes VCL Help

TApplication in the VCL provides four methods that are accessible from application code:

Help methods in TApplication

Method	Description
HelpCommand	Takes a Windows Help style HELP_COMMAND and passes it off to WinHelp. Help requests forwarded through this mechanism are passed only to implementations of IspecialWinHelpViewer.
HelpContext	Invokes the Help System with a request for context-based Help.
HelpKeyword	Invokes the HelpSystem with a request for keyword-based Help.
HelpJump	Requests the display of a particular topic.

All four functions take the data passed to them and forward it through a data member of *TApplication*, which represents the Help system. That data member is directly accessible through the property *HelpSystem*.

How VCL Controls Process Help

All VCL controls that derive from *TControl* expose several properties that are used by the Help system: *HelpType*, *HelpContext*, and *HelpKeyword*.

The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, that can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of Help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWinControl* calls *InvokeHelp*.

Calling a Help System Directly

For additional Help system functionality not provided by VCL or CLX applications, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

Using IHelpSystem

IHelpSystem allows an application to do three things:

- Provides path information to the Help Manager.
- Provides a new Help selector.
- Asks the Help Manager to display Help.

Providing path information is important because the Help Manager is platform-independent and Help system-independent and so is not able to ascertain the location of Help files. If an application expects Help to be provided by an external Help system that is not able to ascertain file locations itself, it must provide this information through the *IHelpSystem's ProvideHelpPath* method, which allows the information to become available through the *IHelpManager's GetHelpPath* method. (This information propagates outward only if the Help viewer asks for it.)

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide Help for the same keyword. For more information, see the topic Implementing IHelpSelector.

IHelpSystem exports four procedures and one function to request the Help Manager to display Help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

Hook is intended entirely for WinHelp compatibility and should not be used in a CLX application; it allows processing of WM_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

Customizing the IDE Help System

The IDE supports multiple Help viewers in exactly the same way that a VCL or CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that the VCL uses.

The IDE comes with two Help viewers installed: the HyperHelp viewer, which allows Help requests to be forwarded to HyperHelp, an external WinHelp emulator under which the Kylix Help files are viewed, and the Man page viewer, which allows you to access the Man system installed on most Unix machines. Because it is necessary for Kylix Help to work, the HyperHelp viewer may not be removed; the Man page viewer ships in a separate package whose source is available in the examples directory.

To install a new Help viewer in the IDE, you do exactly what you would do in a VCL or CLX application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE:

1. Make sure that the unit implementing the Help viewer contains HelpIntfs.pas.
2. Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)

3 Make sure that the Help viewer exists as a global instance within the unit.

4 In the initialization section of the unit, make sure that the instance is passed to the *RegisterHelpViewer* function.

Developing the application user interface

Developing the Application User Interface: Overview

When you open the IDE or create a new project, a blank form is displayed on the screen. You design your application's user interface (UI) by placing and arranging visual components, such as windows, menus, and dialog boxes, from the **Tool palette** onto the form.

Once a visual component is on the form, you can adjust its position, size, and other design-time properties, and code its event handlers. The form takes care of the underlying programming details.

The following topics describe some of the major interface tasks, such as working with forms, creating component templates, adding dialog boxes, and organizing actions for menus and toolbars.

Controlling Application Behavior

TApplication, TScreen, and TForm are the classes that form the backbone of all applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

Working at the Application Level

The global variable *Application*, of type *TApplication*, is in every VCL- or CLX-based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* handles how you call a Help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the **Project** ▸ **Options** Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnMessage* event lets you intercept Windows messages (on Windows only), the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

Handling the Screen

A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying:

- The look of the cursor.
- The size of the window in which your application is running.
- A list of fonts available to the screen device.
- Multiple screen behavior (Windows only).

If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

For CLX applications, the default behavior is that applications create a screen component based on information about the current screen device and assign it to *Screen*.

Using the Main Form

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form:

1. Choose **Project** ▸ **Options** and select the Forms page.
2. In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

Hiding the Main Form

You can prevent the main form from appearing when your application starts by using the global Application variable.

To hide the main form at startup:

1. Choose **Project** ▸ **View Source** to display the main project file.
2. Add the following code after the call to Application.CreateForm and before the call to Application.Run.

```
[Delphi]
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

```
[C++]
Application->ShowMainForm = false;
```

Note: You can set the form's *Visible* property to *False* using the **Object Inspector** at design time rather than setting it at runtime as in the previous example.

Adding Forms

To add a form to your project, select **File** ▸ **New** ▸ **Form**. You can see all your project's forms and their associated units listed in the **Project Manager** (**View** ▸ **Project Manager**) and you can display a list of the forms alone by choosing **View** ▸ **Forms**.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form:

1. Select the form that needs to refer to another.
2. Choose **File** ▸ **Use Unit**.
3. Select the name of the form unit for the form to be referenced.
4. Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the **File** ▸ **Use Unit** command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This generates the "Circular reference" error at compile time.

Managing Layout

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

TControl introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
[Delphi]
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
```

```
[C++]
void __fastcall (__closure *TConstrainedResizeEvent)(System::TObject* Sender, int
&MinWidth, int &MinHeight, int &MaxWidth, int &MaxHeight);
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as var parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBox*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

Using Forms

When you create a form from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

Controlling When Forms Reside in Memory

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:

```
[Delphi]
Application.CreateForm(TForm1, Form1);
```

```
[C++]
Application ->CreateForm(__classid(TForm1), &Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

Displaying an Auto-created Form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
[Delphi]
procedure TMainForm.Button1Click(Sender: TObject);
begin
ResultsForm.ShowModal;
end;
```

```
[C++]
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
ResultsForm->ShowModal();
}
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

Creating Forms Dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE:

1. Select the **File** ▸ **New** ▸ **Form** from the main menu to display the new form.
2. Remove the form from the Auto-create forms list of the **Project** ▸ **Options** ▸ **Forms** page.

   This removes the form's invocation at startup. As an alternative, you can manually remove the following line from program's main entry point:

   ```
   [Delphi]
   Application.CreateForm(TResultsForm, ResultsForm);
   ```

   ```
   [C++]
   Application->CreateForm(__classid(TResultsForm), &ResultsForm);
   ```

3 Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
[Delphi]
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm := TResultForm.Create(self);
  try
    ResultsForm.ShowModal;
  finally
    ResultsForm.Free;
  end;
end;
```

```
[C++]
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
ResultsForm = new TResultsForm(this);
ResultsForm->ShowModal();
delete ResultsForm;
}
```

In the above example, note the use of **try..finally**. Putting in the line ResultsForm.Free; in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

Note: If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the **Project** ▸ **Options** ▸ **Forms** page. Specifically, if you create the new form without deleting the form of the same name from the list, Delphi creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

Creating Modeless Forms Such as Windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

Creating a Form Instance Using a Local Variable

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```
[Delphi]
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

```
[C++]
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
TResultsForm *rf = new TResultsForm(this);
rf->ShowModal();
delete rf;
}
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

Passing Additional Arguments to Forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```
[Delphi]
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

```
[C++]
class TResultsForm : public TForm
{
__published:    // IDE-managed Components
TLabel *ResultsLabel;
TButton *OKButton;
void __fastcall OKButtonClick(TObject *Sender);
private:        // User declarations
```

```
public:         // User declarations
virtual __fastcall TResultsForm(TComponent* Owner);
   virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);
};
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```
[Delphi]
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  inherited Create(Owner);
  case whichButton of
    1: ResultsLabel.Caption := "You picked the first button.";
    2: ResultsLabel.Caption := "You picked the second button.";
    3: ResultsLabel.Caption := "You picked the third button.";
  end;
end;
```

```
[C++]
void__fastcall TResultsForm::TResultsForm(int whichButton, TComponent* Owner)
: TForm(Owner)
{
switch (whichButton) {
case 1:
ResultsLabel->Caption = "You picked the first button!";
break;
case 2:
ResultsLabel->Caption = "You picked the second button!";
break;
case 3:
ResultsLabel->Caption = "You picked the third button!";
}
}
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```
[Delphi]
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
```

```
[C++]
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
TResultsForm *rf = new TResultsForm(2, this);
rf->ShowModal();
delete rf;
}
```

Retrieving Data from Forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

Retrieving Data from Modeless Forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red," "Green," "Blue," and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```
[Delphi]
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

```
[C++]
class TColorForm : public TForm
{
__published:    // IDE-managed Components
TListBox *ColorListBox;
void __fastcall ColorListBoxClick(TObject *Sender);
private:        // User declarations
String getColor();
void   setColor(String);
String curColor;
public:         // User declarations
virtual __fastcall TColorForm(TComponent* Owner);
__property String CurrentColor = {read=getColor, write=setColor};
};
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor*, to store the actual value for the property in the private data member *FColor*:

```
[Delphi]
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

```
[C++]
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
int index = ColorListBox->ItemIndex;
if (index >= 0) {
CurrentColor = ColorListBox->Items->Strings[index];
}
else
CurrentColor = "";
}
//---------------------------------------------------------------------------
void TColorForm::setColor(String s)
{
curColor = s;
}
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton*) on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
[Delphi]
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
```

```
[C++]
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
if (ColorForm) {
String s = ColorForm->CurrentColor;
// do something with the color name string
}
}
```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm's CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, `MainColor := ColorForm.GetColor;`). In fact, there's nothing to prevent another form from getting the ColorForm's currently selected color by checking the listbox selection directly:

```
[Delphi]
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
```

```
[C++]
String TColorForm::getColor()
{
```

```
return curColor;
}
```

```
[C++]
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

Retrieving Data from Modal Forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is when form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
[Delphi]
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

```
[C++]
class TColorForm : public TForm
{
__published:    // IDE-managed Components
TListBox *ColorListBox;
TButton *SelectButton;
TButton *CancelButton;
void __fastcall CancelButtonClick(TObject *Sender);
void __fastcall SelectButtonClick(TObject *Sender);
private:        // User declarations
String* curColor;
public:         // User declarations
virtual __fastcall TColorForm(TComponent* Owner);
virtual __fastcall TColorForm(String* s, TComponent* Owner);
};
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
[Delphi]
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

```
[C++]
void__fastcall TColorForm::TColorForm(String* s, TComponent* Owner)
: TForm(Owner)
{
curColor = s;
*curColor = "";
}
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

Note: To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see Controlling when forms reside in memory.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
[Delphi]
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

```
[C++]
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
int index = ColorListBox->ItemIndex;
if (index >= 0)
*curColor = ColorListBox->Items->Strings[index];
Close();
}
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would look as follows:

```
[Delphi]
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
```

```
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;
procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

```
[C++]
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
String s;
GetColor(&s);
if (s != "") {
// do something with the color name string
}
else {
// do something else because no color was picked
}
}
//---------------------------------------------------------------------
void TResultsForm::GetColor(String *s)
{
ColorForm = new TColorForm(s, this);
ColorForm->ShowModal();
delete ColorForm;
ColorForm = 0; // NULL the pointer
}
```

UpdateButtonClick creates a String called MainColor. The address of MainColor is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor, assuming that a color was selected. Otherwise, MainColor contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor default to an empty string).

Reusing Components and Groups of Components

You can save and reuse work you've done with components using several tools:

- Configure and save groups of components in component templates.
- Save forms, data modules, and projects in the Object Repository. The Repository gives you a central database of reusable elements and lets you use form inheritance to propagate changes.
- Save frames on the **Tool palette** or in the Repository. Frames use form inheritance and can be embedded into forms or other frames.

- Create a custom component, the most complicated but most flexible way of reusing code. See Overview of Component Creation.

Creating and Using Component Templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the **Tool palette**, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template:

1. Place and arrange components on a form. In the **Object Inspector**, set their properties and events as desired.
2. Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.
3. Choose **Component** ▸ **Create Component Template**.
4. Specify a name for the component template in the Component Template Information edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template." For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate." You can change this name, but be careful not to duplicate existing component names.
5. In the Palette page edit box, specify the **Tool palette** page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.
6. Next to Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.
7. Click OK.

To remove templates from the **Tool palette**, choose **Component** ▸ **Configure Palette**.

Working with Frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the **Tool palette** for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

Creating frames

Using and modifying frames

Creating Frames

To create an empty frame, choose **File** ▸ **New** ▸ **Frame**, or choose **File** ▸ **New** ▸ **Other** and double-click Frame. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose **File** ▸ **New** ▸ **Application**, close the new form and unit without saving them, then choose **File** ▸ **New** ▸ **Frame** and save the project.

Note: When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing **View** ▸ **Forms** and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's form file by right-clicking and choosing View as Form or View as Text.

Adding frames to the Tool palette

Frames are added to the **Tool palette** as component templates. To add a frame to the **Tool palette**, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

Using and Modifying Frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

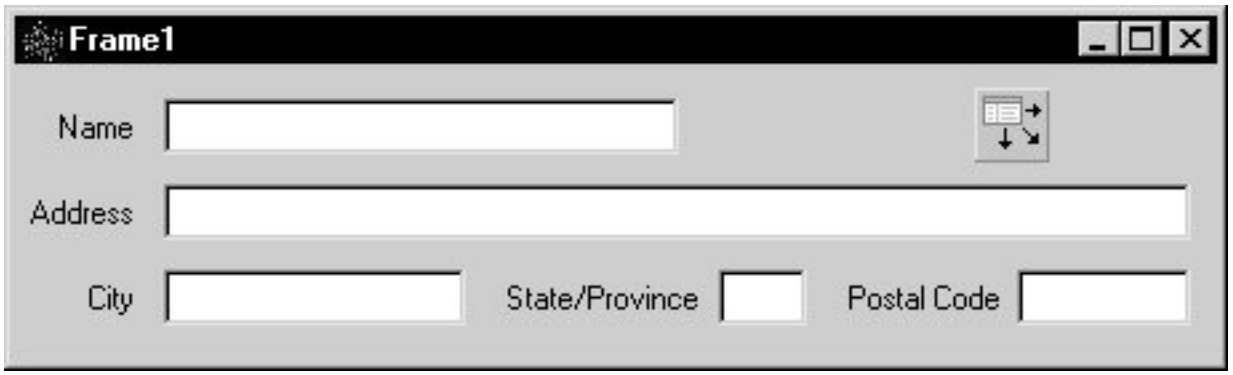
- Select a frame from the **Tool palette** and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.
- Select *Frames* from the Standard category of the **Tool palette** and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

A frame with data-aware controls and a data source component:



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the Picture property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Image* object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

Sharing Frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas) and form (.dfm or .xfm) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see Using the Object Repository.

If you send a frame's unit and form files to other developers, they can open them and add them to the **Tool palette**. If the frame has other frames embedded in it, they will have to open it as part of a project.

Developing Dialog Boxes

The dialog box components on the Dialogs category of the **Tool palette** make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns *False*.

Note: For CLX applications, you can use the dialogs provided in the QDialogs unit. For operating systems that have native dialog box types for common tasks, such as for opening or saving a file or for changing font or color, you can use the *UseNativeDialog* property. Set *UseNativeDialog* to *True* if your application will run in such an environment, and if you want it to use the native dialogs instead of the Qt dialogs.

Using Windows Common Dialog Boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user's file is stored in the *TOpenDialog FileName* property, which you can then process as you want.

The following code can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem's *OnClick* event:

```
[Delphi]
if OpenDialog1.Execute then
   filename := OpenDialog1.FileName;
```

```
[C++]
if(OpenDialog1->Execute()){
   filename = OpenDialog1->FileName;
};
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named filename.

Organizing Actions for Toolbars and Menus

Several features simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

On the Windows platform, tools are provided to make it easy to define and group actions, create different layouts, and customize menus at design time or runtime. These tools are known collectively as ActionBand tools, and the menus and toolbars you create with them are known as action bands. In general, you can create an ActionBand user interface as follows:

- Build the action list to create a set of actions that will be available for your application (use the Action Manager, TActionManager)
- Add the user interface elements to the application (use ActionBand components such as TActionMainMenuBarand TActionToolBar)
- Drag-and-drop actions from the Action Manager onto the user interface elements

The following table defines the terminology related to setting up menus and toolbars:

Action setup terminology

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File SaveAs, File Run, and File Exit are included along with many others for editing, formatting,

	searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists and the Action Manager.
Action band	A container for a set of actions associated with a customizable menu or toolbar. The ActionBand components for main menus and toolbars (*TActionMainMenuBar* and *TActionToolBar*) are examples of action bands.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as *TEditCopy*, *TEditCut*, and *TEditUndo*. You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Action Manager	Groups and organizes logical sets of actions that can be reused on ActionBand components. See *TActionManager*.
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using the ActionBand menu class *TActionMainMenuBar*, or by using cross-platform components such as *TMainMenu* or *TPopupMenu*.
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the ActionBand toolbar component *TActionToolBar*, or by using the cross-platform component *TToolBar*.

If you are doing cross-platform development, refer to Using action lists for details.

What Is an Action?

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, **Tools** ▸ **Customize**).

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox, TRadioButton*, and so on). Actions also reside on ActionBand components such as TActionMainMenuBar *and* TActionToolBar. When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client's *OnClick* event is associated with its action's *OnExecute* event.
- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to *False*), every client of that action is disabled, appearing grayed.

You can add, delete, and rearrange actions using the Action Manager or the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls. See Creating toolbars and menus and, for cross-platform development, Setting up action lists for details.

Setting Up Action Bands

Because actions do not maintain any "layout" (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus.

You organize sets of actions using the Action Manager (TActionManager). You can use standard actions provided or create new actions of your own.

You then create the action bands:

- Use TActionMainMenuBar to create a main menu.
- Use TActionToolBarto create a toolbar.

The action bands act as containers that hold and render sets of actions. You can drag and drop items from the Action Manager editor onto the action band at design time. At runtime, application users can also customize the application's menus or toolbars using a dialog box similar to the Action Manager editor.

Creating Toolbars and Menus

Note: This topic describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu,* organizing them using action lists (*TActionList*). See Setting up action lists for details.

You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu items or as buttons on a toolbar.

The general procedure for creating menus, toolbars, and other action bands involves these steps:

- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

The following procedure explains these steps in more detail.

To create menus and toolbars using action bands:

1. From the Additional category of the **Tool palette**, drop an Action Manager component (TActionManager) onto the form where you want to create the toolbar or menu.
2. If you want images on the menu or toolbar, drop an ImageList component from the Win32 category of the **Tool palette** onto a form. (You need to add the images you want to use to the ImageList or use the one provided.)
3. From the Additional category of the **Tool palette**, drop one or more of the following action bands onto the form:

- TCustomActionMainMenuBar(for designing main menus)

- TActionToolBar(for designing toolbars)

4. Connect the ImageList to the Action Manager: with focus on the Action Manager and in the **Object Inspector**, select the name of the ImageList from the Images property.
5. Add actions to the Action Manager editor's action pane:
   - Double-click the Action Manager to display the Action Manager editor.
   - Click the drop-down arrow next to the New Action button (the leftmost button at the top right corner of the Actions tab) and select New Action or New Standard Action. A tree view is displayed. Add one or more actions or categories of actions to the Action Manager's actions pane. The Action Manager adds the actions to its action lists.
6. Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar you are designing.

To add user-defined actions, create a new *TAction* by pressing the New Action button and writing an event handler that defines how it will respond when fired. See What happens when an action fires for details. Once you've defined the actions, you can drag and drop them onto menus or toolbars like the standard actions.

Adding Color, Patterns, or Pictures to Menus, Buttons, and Toolbars

You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner the runs up the left or right side of a menu.

You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign a color, pattern, or bitmap in the *Background* property in the **Object Inspector**.

Use the *BackgroundLayout* property to describe how to place the background on the element. Colors or images can be placed behind the caption normally, stretched to fit the item area, or tiled in small squares to cover the area.

Items with normal (blNormal), stretched (blStretch), or tiled (blTile) backgrounds are rendered with a transparent background. If you create a banner, the full image is placed on the left (blLeftBanner) or the right (blRightBanner) of the item. You need to make sure it is the correct size because it is not stretched or shrunk to fit.

To change the background of an action band (that is, on a main menu or toolbar), select the action band and choose the *TActionClientBar* through the action band collection editor. You can set *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on the entire toolbar or menu.

Adding Icons to Menus and Toolbars

You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an *ImageList* component.

To add icons to menus and toolbars:

1. Drop an *ImageList* component from the Win32 category of the **Tool palette** onto a form.
2. Add the images you want to use to the image list: Double-click the *ImageList* icon. Click Add and navigate to the images you want to use and click OK when done. Some sample images are included in Program Files\Common Files\Borland Shared\Images. (The buttons images include two views of each for active and inactive buttons.)
3. From the Additional category of the **Tool palette**, drop one or more of the following action bands onto the form:
   - TActionMainMenuBar(for designing main menus)

- TActionToolBar *(for designing* toolbars)

4. Connect the image list to the Action Manager. First, set the focus on the Action Manager. Next, in the **Object Inspector**, select the name of the image list from the *Images* property, such as ImageList1.
5. Use the Action Manager editor to add actions to the Action Manager. You can associate an image with an action by setting its *ImageIndex* property to its number in the image list.
6. Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar.
7. For toolbars where you only want to display the icon and no caption: select the Toolbar action band and double-click its Items property. In the collection editor, you can select one or more items and set their *Caption* properties.
8. The images automatically appear on the menu or toolbar.

Selecting Menu and Toolbar Styles

Just as you can add different colors and icons to individual menus and toolbars, you can select different menu and toolbar styles to give your application a comprehensive look and feel. In addition to the standard style, your application can take on the look of Windows XP, Encarta™, or a custom presentation using a coordinated color scheme. To give your application a coherent look and feel, the IDE uses colormaps.

A colormap can be simple, merely adding the appropriate colors to existing menus and toolbars. Or, a colormap can be complex, altering numerous subtle details of a menu's or toolbar's look and feel, including the smallest button edges or menu shadows. The XP colormap, for example, has numerous subtle refinements for menu and toolbar classes. The IDE handles the details for you, automatically using the appropriate colormaps.

By default, the component library uses the XP style. To centrally select an alternate style for all your application's menus and toolbars, use the *Style* property on the *ActionManager* component.

To select menu and toolbar styles:

1. From the Additional category of the **Tool palette**, drop an *ActionManager* component onto a form.
2. In the **Object Inspector**, select the *Style* property. You can choose from a number of different styles.
3. Once you've selected a style, your application's menus and toolbars will take on the look of the new colormap.

You can customize the look and feel of a style using colormap components.

To customize the look and feel of a colormap:

1. From the Additional category of the **Tool palette**, drop the appropriate colormap component onto a form (for example, *XPColorMap* or *StandardColorMap*). In the **Object Inspector**, you will see numerous properties to adjust appearance, many with drop downs from which you can select alternate values.
2. Change each ToolBar or menu's *ColorMap* property to point to the colormap object that you dropped on the form.
3. In the **Object Inspector**, adjust the colormap's properties to change the appearance of your toolbars and menus as desired.

Note: Be careful when customizing a colormap. When you select a new, alternate colormap, your old settings will be lost. You may want to save a copy of your application if you want to experiment with alternate settings and possibly return to a previous customization.

Creating Dynamic Menus

Dynamic menus and toolbars allow users to modify the application in various ways at run time. Some examples of dynamic usage include customizing the appearance of toolbars and menus, hiding unused items, and responding to most recently used lists (MRUs).

Creating Customizable Toolbars and Menus

You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.

To allow the user of your application to customize an action band in your application:

1. Drop an Action Manager component onto a form.
2. Drop your action band components (TCustomActionMainMenuBar,TActionToolBar).
3. Double-click the Action Manager to display the Action Manager editor:
   - Add the actions you want to use in your application. Also add the Customize action, which appears at the bottom of the standard actions list.
   - Drop a *TCustomizeDlg* component from the Additional tab onto the form, and connect it to the Action Manager using its ActionManager property. You specify a filename for where to stream customizations made by users.
   - Drag and drop the actions onto the action band components. (Make sure you add the Customize action to the toolbar or menu.)
4. Complete your application.

When you compile and run the application, users can access a Customize command that displays a customization dialog box similar to the Action Manager editor. They can drag and drop menu items and create toolbars using the same actions you supplied in the Action Manager.

Hiding Unused Items and Categories in Action Bands

One benefit of using ActionBands is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a drop-down button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in a particular collection (like the File menu), or all of the items in a given action band.

The action manager keeps track of the number of times an action has been called by the user, which is stored in the associated *TActionClientItem's UsageCount* field. The action manager also records the number of times the application has been run, which we shall call the session number, as well as the session number of the last time an action was used. The value of *UsageCount* is used to look up the maximum number of sessions the item can go unused before it becomes hidden, which is then compared with the difference between the current session number and the session number of the last use of the item. If that difference is greater than the number determined in *PrioritySchedule*, the item is hidden. The default values of *PrioritySchedule* are shown in the table below:

Default values of the action manager's PrioritySchedule property

Number of sessions in which an action band item was used	Number of sessions an item will remain unhidden after its last use
0, 1	3

2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 or more	31

It is possible to disable item hiding at design time. To prevent a specific action (and all the collections containing it) from becoming hidden, find its *TActionClientItem* object and set its *UsageCount* to -1. To prevent hiding for an entire collection of items, such as the File menu or even the main menu bar, find the *TActionClients* object associated with the collection and set its *HideUnused* property to *False*.

Creating Most Recently Used Lists

A most recently used list (MRU) reflects the user's most recently accessed files in a specific application. Using action bands, you can code MRU lists in your applications.

When building MRUs for your applications, it is important not to hard code references to specific numerical indexes into the Action Manager's *ActionBars* property. At runtime, the user may change the order of items or even delete them from the action bands, which in turn will change the numerical ordering of the index. Instead of referring to index numbering, *TActionManager* includes methods that facilitate finding items by action or by caption.

For more information about MRU lists, sample code, and methods for finding actions in lists, see FindItemByAction and FindItemByCaption in the online Help.

Using Action Lists

Note: The contents of this topic apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See Organizing actions for toolbars and menus for details.

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

Setting Up Action Lists

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

1. Drop a *TActionList* object onto your form or data module. (ActionList is on the Standard category of the **Tool palette**.)
2. Double-click the *TActionList* object to display the Action List editor.
   - Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
   - The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.
   - Or, create a new action of your own: right-click and choose New Action.
3. Set the properties of each action in the **Object Inspector**. (The properties you set affect every client of the action.)

   The Name property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible*, and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's Enabled property corresponds to a *TToolButton's Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton's Down* property.
4. If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See What happens when an action fires for details.
5. Attach the actions in the action list to the clients that require them:
   - Click on the control (such as the button or menu item) on the form or data module. In the **Object Inspector**, the *Action* property lists the available actions.
   - Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired. See What happens when an action fires for details.

What Happens When an Action Fires

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
[Delphi]
procedure TForm1.Action1Execute(Sender: TObject);
begin
{ Toggle Toolbar1's visibility }
ToolBar1.Visible := not ToolBar1.Visible;
end;
```

```
[C++]
void __fastcall TForm1::Action1Execute(TObject *Sender)
{
// Toggle Toolbar1's visibility
ToolBar1->Visible = !ToolBar1->Visible;
}
```

Note: For general information about events and event handlers, see Working with Events and Event Handlers.

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Delphi calls the action's Execute method which defers first to the action list, then the Application object, then the action itself if neither action list nor Application handles it. To explain this in more detail, Delphi follows this dispatching sequence when looking for a way to respond to the user action:

If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

The action list's event handler has a parameter called Handled, that returns False by default. If the handler is assigned and it handles the event, it returns *True*, and the processing sequence ends here. For example:

```
[Delphi]
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
Handled := True;
end;
```

```
[C++]
void __fastcall TForm1::ApplicationExecuteAction(TBasicAction *Action, bool &Handled)
{
  // Prevent execution of all actions in Application
Handled = true;
}
```

```
[C++]
void __fastcall TForm1::ActionList1ExecuteAction(TBasicAction *Action, bool &Handled)
{
Handled = true;
}
```

If you don't set Handled to *True* in the action list event handler, then processing continues.

If you did not write an *OnExecute* event handler for the action list or if the event handler doesn't handle the action, the application's *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

The global Application object receives an OnActionExecute event if any action list in the application fails to handle an event. Like the action list's *OnExecute* event handler, the *OnActionExecute* handler has a parameter Handled that returns *False* by default. If an event handler is assigned and handles the event, it returns *True*, and the processing sequence ends here. For example:

```
[Delphi]
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
{ Prevent execution of all actions in Application }
Handled := True;
end;
```

If the application's *OnExecute* event handler doesn't handle the action, the action's *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See how actions find their targets for details on how the application locates a target that can respond to a predefined action class.

How Actions Find Their Targets

What happens when an action fires describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

1. Active control: The application looks first for an active control as a potential target.
2. Active form: If the application does not find an active control or if the active control can't act as a target, it looks at the screen's ActiveForm.
3. Controls on the form: If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

Updating Actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

```
[Delphi]
procedure TForm1.Action1Update(Sender: TObject);
begin
{ Indicate whether ToolBar1 is currently visible }
(Sender as TAction).Checked := ToolBar1.Visible;
end;
```

```
[C++]
void __fastcall TForm1::Action1Update(TObject *Sender)
{
// Indicate whether ToolBar1 is currently visible
((TAction *)Sender)->Checked = ToolBar1->Visible;
}
```

Warning: Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

Predefined Action Classes

You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects that automatically perform actions. The predefined actions are organized within the following classes:

Class	Description
Edit	Standard edit actions: Used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the clipboard.
Format	Standard formatting actions: Used with rich text to apply text formatting options such as bold, italic, underline, strikeout, and so on. *TRichEditAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement formatting of the target.
Help	Standard Help actions: Used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command onto a Help system.
Window	Standard window actions: Used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
File	File actions: Used with operations on files such as File Open, File Run, or File Exit.
Search	Search actions: Used with search options. *TSearchAction* implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.
Tab	Tab control actions: Used to move between tabs on a tab control such as the Prev and Next buttons on a wizard.
List	List control actions: Used for managing items in a list view.
Dialog	Dialog actions: Used with dialog components. *TDialogAction* implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
Internet	Internet actions: Used for functions such as Internet browsing, downloading, and sending mail.
DataSet	DataSet actions: Used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target. *TDataSetAction* introduces a *DataSource* property that ensures actions are performed on that dataset. If *DataSource* is nil, the currently focused data-aware control is used.
Tools	Tools: Additional tools such as *TCustomizeActionBars* for automatically displaying the customization dialog for action bands.

All of the action objects are described under the action object names in the online Help.

Writing Action Components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of objects. Then, you can use your custom actions in the same way you use predefined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the QStdActns and DBActns units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction, TWindowAction*, and so on) generally override *HandlesTarget*, *UpdateTarget*, and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Methods overriden by base classes of specific actions

Method	Description
HandlesTarget	Called automatically when the user invokes an object (such as a tool button or menu item) that is linked to the action. The *HandlesTarget* method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the *Target* parameter as a "target". See How actions find their targets for details.
UpdateTarget	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of *OnUpdateAction*. See Updating actions for details.
ExecuteTarget	Called automatically when the action fires in response to a user action in place of *OnExecute* (for example, when the user selects a menu item or presses a tool button that is linked to this action). See What happens when an action fires for details.

When you write your own action classes, it is important to understand the following:

- How actions find their targets
- Registering actions

Registering Actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:

```
[Delphi]
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

```
[C++]
extern PACKAGE void __fastcall UnRegisterActions(TMetaClass* const * AClasses, const int
AClasses_Size);
```

```
[C++]
extern PACKAGE void __fastcall RegisterActions(const AnsiString CategoryName, TMetaClass*
const * AClasses, const int AClasses_Size, TMetaClass* Resource);
```

When you call RegisterActions, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

For example, the following code registers the standard actions with the IDE:

```
[Delphi]
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

```
[C++]
namespace MyAction
{
  void __fastcall PACKAGE Register()
  {
     // code goes here to register any components and editors
TMetaClass classes[2] = {__classid(TMyAction1), __classid(TMyAction2)};
RegisterActions("MySpecialActions", classes, 1, NULL);
}
}
```

When you call UnRegisterActions, the actions no longer appear in the Action List editor.

Creating and Managing Menus

Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This topic explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer.
- Building menus.
- Editing menu items in the Object Inspector.
- Using the Menu Designer context menu.
- Using menu templates.
- Saving a menu as a template.
- Adding images to menu items.

For information about hooking up menu items to the code that executes when they are selected, see .

Opening the Menu Designer

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a *TMainMenu* or *TPopupMenu* component to your form. Both menu components are located on the Standard category of the **Tool palette**.

A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either:

- Double-click the menu component.
- Or, from the Properties page of the **Object Inspector**, select the Items property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the **Object Inspector**.

Building Menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

For more information about menu templates, see Using menu templates.

For more information about creating a menu using the menu designer see

- Naming menus
- Naming the menu items
- Adding, inserting, and deleting menu items
- Creating submenus
- Adding images to menu items
- Viewing the menu

Naming Menus

As with all components, when you add a menu component to the form, the form gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows language naming conventions.

The menu name is added to the form's type declaration, and the menu name then appears in the Component list.

Naming the Menu Items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption.

For example, if you give a menu item a *Caption* property value of File, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank until you type a value.

Note: If you enter characters in the *Caption* property that are not valid for Delphi identifiers, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Delphi adds any menu item names to the form's type declaration, and those names then appear in the Component list.

Adding, Inserting, and Deleting Menu Items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time:

1. Select the position where you want to create the menu item.
2. If you've just opened the Menu Designer, the first position on the menu bar is already selected.

   Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the **Object Inspector** and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.
3. Press Enter.

   The next placeholder for a menu item is selected.

   If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Delphi has filled in the *Name* property based on the value you entered for the caption. (See Naming the menu items.)
4. Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press Esc to return to the menu bar.

   Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press Enter to complete an action. To return to the menu bar, press Esc .

To insert a new, blank menu item:

1. Place the cursor on a menu item.
2. Press Ins.

Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command:

1. Place the cursor on the menu item you want to delete.
2. Press Del.

Note: You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

Separator bars insert a line between menu items and items on a toolbar. You can use separator bars to indicate groupings within the menu list or toolbar, or simply to provide a visual break in a list.

To add a separator bar to a menu:

- Add a menu item as described above and type a hyphen (-) for the caption.
- Or press the hyphen (-) key while the cursor is positioned on the menu where you want a separator to appear.

To add a separator bar onto a TActionToolBar, press the insert key and set the new item's caption to a separtor bar (|) or hyphen (-).

To add accelerators or shortcuts to menu items, see Specifying accelerator keys and keyboard shortcuts.

Specifying Accelerator Keys and Keyboard Shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual* **.**

To specify an accelerator, add an ampersand in front of the appropriate letter. For example, to add a Save menu command with the S as an accelerator key, type &Save.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut, use the **Object Inspector** to enter a value for the ShortCut property, or select a key combination from the drop-down list. This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

Warning: Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

Creating Submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

To create a submenu:

1. Select the menu item under which you want to create a submenu.
2. Press Ctrl+RIGHT ARROW to create the first placeholder, or right-click and choose Create Submenu.
3. Type a name for the submenu item, or drag an existing menu item into this placeholder.
4. Press ENTER, or DOWN ARROW, to create the next placeholder.
5. Repeat steps 3 and 4 for each item you want to create in the submenu.
6. Press ESC to return to the previous menu level.

Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well. Moving a menu item into an existing submenu just creates one more level of nesting.

Moving Menu Items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a *different* menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a different menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar:

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
2. Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list:

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.
2. Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Adding Images to Menu Items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image

list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add an image to a menu item using an image list:

1. Drop a *TMainMenu* or *TPopupMenu* object on a form.
2. Drop a *TImageList* object on the form.
3. Open the ImageList editor by double clicking on the *TImageList* object.
4. Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.
5. Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.
6. Create your menu items and submenu items as described in this topic group.
7. Select the menu item you want to have an image in the **Object Inspector** and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

Note: Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

Viewing the Menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu:

1. If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.
2. If the form has more than one menu, select the menu you want to view from the form's Menu property drop-down list.

   The menu appears in the form exactly as it will when you run the program.

Editing Menu Items in the Object Inspector

This topic has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the **Object Inspector**, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the **Object Inspector**. You can switch focus to the **Object Inspector** and continue editing the menu item properties there. Or you can select the menu item from the Component list in the **Object Inspector** and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items:

1. Switch focus from the Menu Designer window to the **Object Inspector** by clicking the properties page of the **Object Inspector**.
2. Close the Menu Designer as you normally would.

   The focus remains in the **Object Inspector**, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

Using the Menu Designer Context Menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to Using menu templates.)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose a .rc or .mnu file to open in the current form.

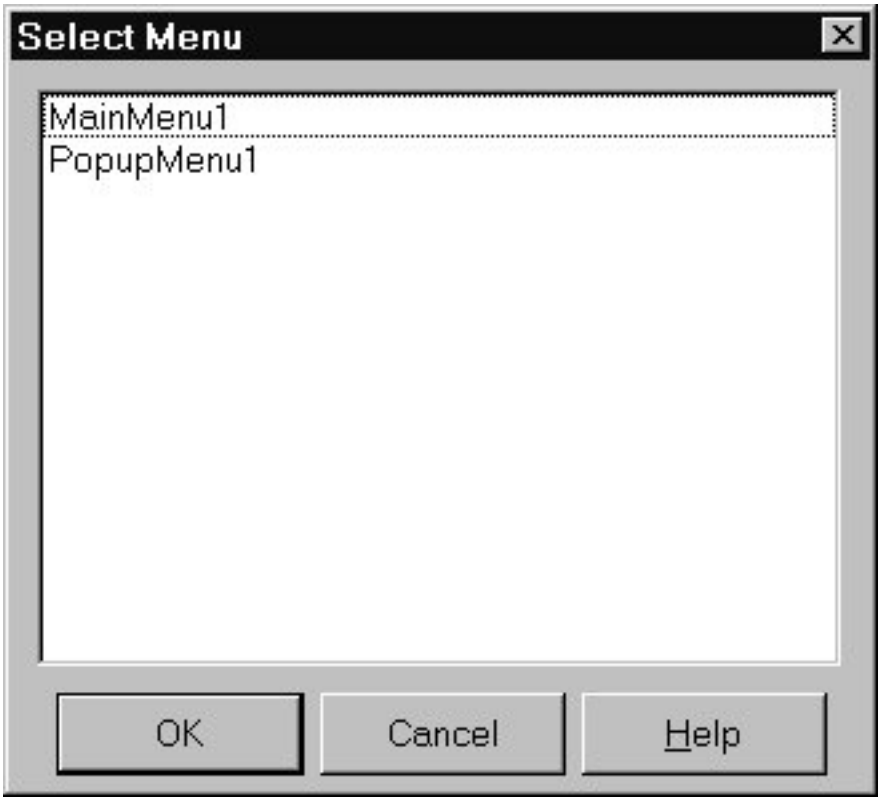
Switching Between Menus at Design Time

If you're designing several menus for your form, you can use the Menu Designer context menu or the **Object Inspector** to easily select and move among them.

To use the context menu to switch between menus in a form:

1. Right-click in the Menu Designer and choose Select Menu.

   The Select Menu dialog box appears.



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2 From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form:

1 Give focus to the form whose menus you want to choose from.

2 From the Component list, select the menu you want to edit.

3 On the Properties page of the **Object Inspector**, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

Using Menu Templates

Several predesigned menus, or menu templates, contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates are stored in the BIN subdirectory in a default installation and have a .dmt extension.

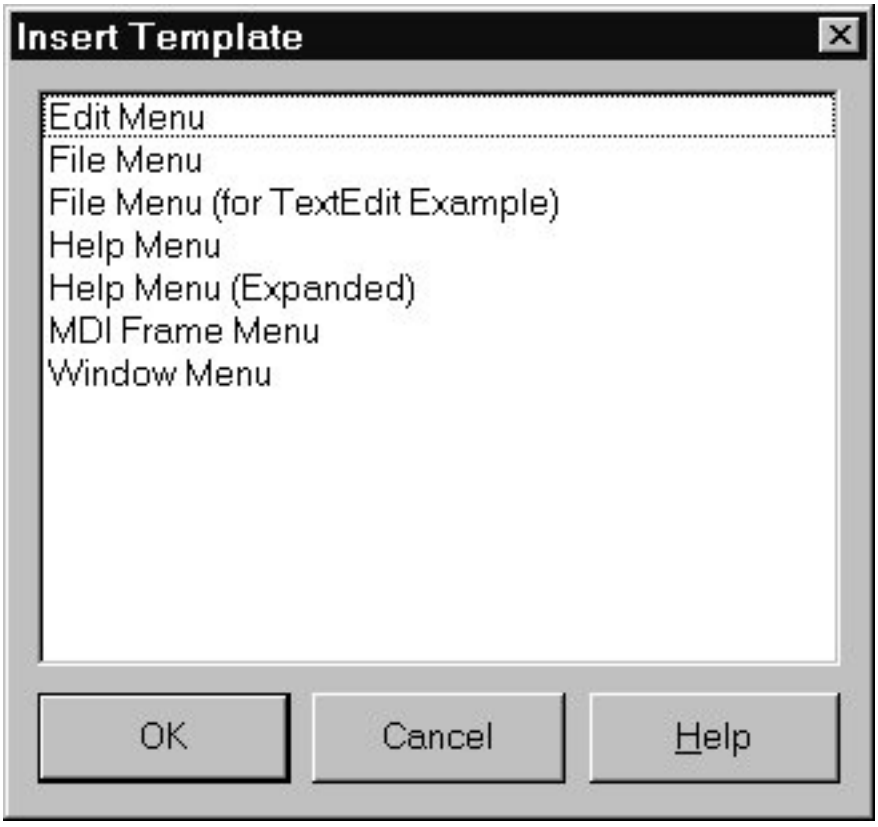
You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application

1 Right-click the Menu Designer and choose Insert From Template.

(If there are no templates, the Insert From Template option appears dimmed in the context menu.)

The Insert Template dialog box opens, displaying a list of available menu templates.



2 Select the menu template you want to insert, then press Enter or choose OK.

This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template

1 Right-click the Menu Designer and choose Delete Templates.

(If there are no templates, the Delete Templates option appears dimmed in the context menu.)

The Delete Templates dialog box opens, displaying a list of available templates.

2 Select the menu template you want to delete, and press Del.

Delphi deletes the template from the templates list and from your hard disk.

Saving a Menu as a Template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in your BIN subdirectory as .dmt files.

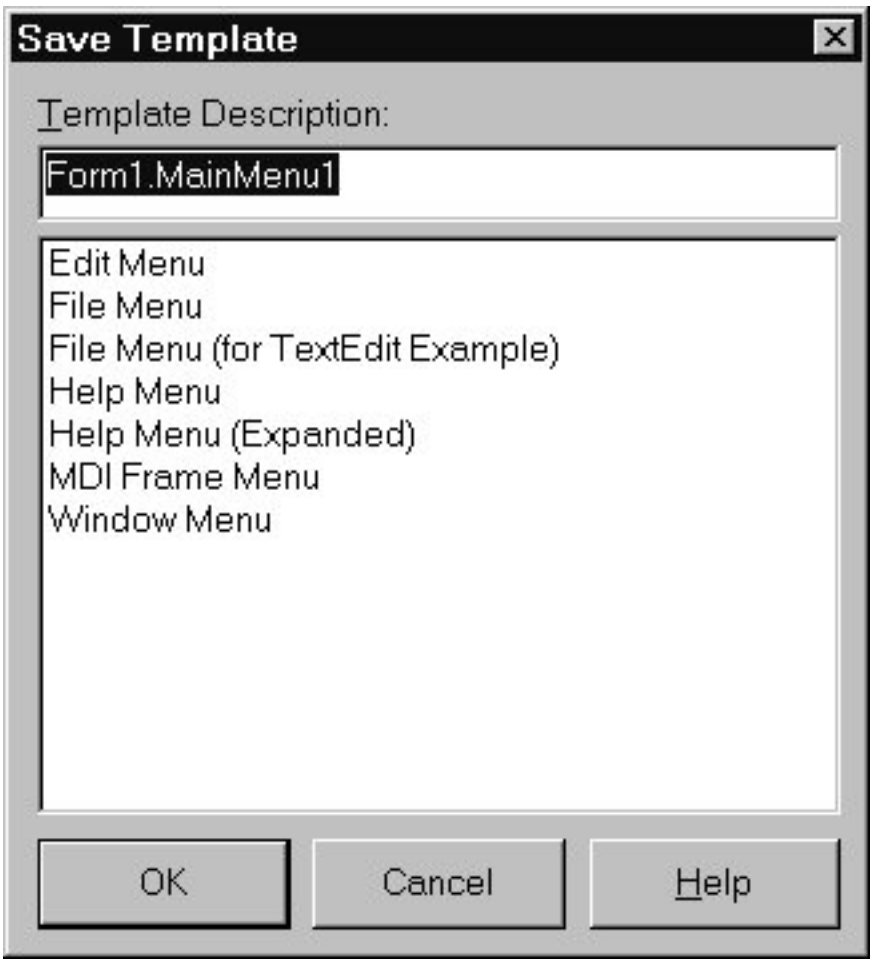
To save a menu as a template

1. Design the menu you want to be able to reuse.

   This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

2. Right-click in the Menu Designer and choose Save As Template.

   The Save Template dialog box appears.



3. In the Template Description edit box, type a brief description for this menu, and then choose OK.

   The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

Note: The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

Naming Conventions for Template Menu Items and Event Handlers

When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert it into a menu with an existing menu item named File1, Delphi names it *File2*.

Delphi also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Delphi still generates the event handler name. You can easily associate items in the menu template with existing *OnClick* event handlers in the form.

For more information, see Associating menu events with event handlers.

Manipulating Menu Items at Runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the Enabled property.

For examples that use the menu item's *Visible* and *Enabled* properties, see Disabling menu items.

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See Merging menus for more information.

Merging Menus

For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

Specifying the Active Menu: Menu Property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
[Delphi]
Form1.Menu := SecondMenu;
```

```
[C++]
Form1->Menu = SecondMenu;
```

Determining the Order of Merged Menu Items: GroupIndex Property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

Rules	Description
Lower numbers appear first (farther left) in the menu.	For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.
To replace items in the main menu, give items on the child menu the same *GroupIndex* value.	This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well. Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.
To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.	For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Importing Resource Files

You can build menus with other applications, so long as the menus are in the standard Windows resource (.RC) file format. You can import such menus directly into your project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files

1. In the Menu Designer, place your cursor where you want the menu to appear.

   The imported menu can be part of a menu you are designing, or an entire menu in itself.
2. Right-click and choose Insert From Resource.

   The Insert Menu From Resource dialog box appears.
3. In the dialog box, select the resource file you want to load, and choose OK.

   The menu appears in the Menu Designer window.

Note: If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

Designing Toolbars and Cool Bars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

Note: Cool bars are not available in CLX applications.

You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

You can add a toolbar to a form in several ways:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.
- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.
- Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to:

- Adding a toolbar using a panel component.
- Adding a toolbar using the toolbar component.
- Adding a cool bar component.
- Responding to clicks.
- Adding hidden toolbars.
- Hiding and showing toolbars.

Adding a Toolbar Using a Panel Component

To add a toolbar to a form using the panel component

1. Add a panel component to the form (from the Standard category of the **Tool palette**).
2. Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
3. Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Adding a speed button to a panel.
- Assigning a speed button's glyph.
- Setting the initial condition of a speed button.
- Creating a group of speed buttons.

- Allowing toggle buttons.

Adding a Speed Button to a Panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional category of the **Tool palette**) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

Assigning a Speed Button's Glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time

1. Select the speed button.
2. In the Object Inspector, select the *Glyph* property.
3. Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

Setting the Initial Condition of a Speed Button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

The table below lists some actions you can set to change a speed button's appearance:

Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	*GroupIndex* property to a value other than zero and its *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*I*ndent property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

Creating a Group of Speed Buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

Allowing Toggle Buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a toggle. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

Adding a Toolbar Using the Toolbar Component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not.

To add a toolbar to a form using the toolbar component

1. Add a toolbar component to the form (from the Win32/Common Controls category of the **Tool palette**). The toolbar automatically aligns to the top of the form.
2. Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can:

- Act like regular pushbuttons.
- Toggle on and off when clicked.
- Act like a set of radio buttons.

To implement tool buttons on a toolbar, do the following:

- Adding a tool button
- Assigning images to tool buttons
- Setting tool button appearance and initial conditions
- Creating groups of tool buttons
- Allowing toggled tool buttons

Adding a Tool Button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the **Tool palette** onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

Assigning Images to Tool Buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled.

To assign images to tool buttons at design time

1. Select the toolbar on which the buttons appear.
2. In the **Object Inspector**, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.
3. Select a tool button.
4. In the **Object Inspector**, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

Setting Tool Button Appearance and Initial Conditions

The table below lists some actions you can set to change a tool button's appearance:

Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on tool button) *Style* property to *tbsCheck* and *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*I*ndent property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	*Flat* property to *True*.

Note: Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to True.

To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to False.

Creating Groups of Tool Buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

Allowing Toggled Tool Buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.

Adding a Cool Bar Component

Note: The *TCoolBar* component requires version 4.70 or later of COMCTL32.DLL and is not available in CLX applications.

The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form in a VCL application:

1. Add a cool bar component to the form (from the Win32 page of the **Tool palette**). The cool bar automatically aligns to the top of the form.
2. Add windowed controls from the **Tool palette** to the bar.

Only VCL components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to a cool bar, but they will not appear on separate bands.

Setting the Appearance of the Cool Bar

The cool bar component offers several useful configuration options. The table below lists some actions you can set to change a tool button's appearance:

Setting a cool button's appearance

To make the cool bar:	Set the toolbar's:
Resize automatically to accommodate the bands it contains	*AutoSize* property to *True* .
Bands maintain a uniform height	*FixedSize* property to *True*.
Reorient to vertical rather than horizontal	*Vertical* property to *True*. This changes the effect of the *FixedSize* property.
Prevent the *Text* properties of the bands from displaying at runtime	*ShowText* property to *False*. Each band in a cool bar has its own *Text* property.
Remove the border around the bar	*BandBorderStyle* to *bsNone.*
Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)	*FixedOrder* to True.
Create a background image for the cool bar	*Bitmap* property to *TBitmap* object.
Choose a list of images to appear on the left of any band	*Images* property to *TImageList* object.

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the **Object Inspector**. Then select a band and assign a value to its *ImageIndex* property.

Note: The cool bar component is not available in CLX applications.

Responding to Clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For general information about events and event handlers, see Working with Events and Event Handlers and Generating a handler for a component's default event.

Assigning a Menu to a Tool Button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

To assign a menu to a tool button

1. Select the tool button.
2. In the **Object Inspector**, assign a pop-up menu (*TPopupMenu*) to the tool button's *DropDownMenu* property.

If the menu's *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

Adding Hidden Toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar:

1. Add a toolbar, cool bar, or panel component to the form.
2. Set the component's *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

Hiding and Showing Toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To show or hide a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
[Delphi]
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

```
[C++]
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
```

Demo Programs: Actions, Action Lists, Menus, and Toolbars

For examples of Windows applications that use actions, action lists, menus, and toolbars, refer to Program Files \Borland\Delphi9\Demos\RichEdit. In addition, the Application wizard (**File** ▸ **New** ▸ **Other**), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects. For examples of cross-platform applications, refer to Demos\CLX.

Common Controls and XP Themes

Microsoft has forked Windows common controls into two separate versions. Version 5 is available on all Windows versions from Windows 95 or later; it displays controls using a "3D chiseled" look. Version 6 became available with Windows XP. Under version 6, controls are rendered by a theme engine which matches the current Windows XP theme. If the user changes the theme, version 6 common controls will match the new theme automatically. You don't need to recompile the application.

The VCL can now accommodate both types of common controls. Borland has added a number of components to the VCL to handle common control issues automatically and transparently. These components will be present in any VCL application you build. By default, any VCL applications will display version 5 common controls. To display version 6 controls, you (or your application's users) must add a manifest file to your application.

A manifest file contains an XML list of dependencies for your application. The file itself shares the name of your application, with ".manifest" appended to the end. For example, if your project creates Project1.exe as its executable, its manifest file should be named Project1.exe.manifest. Here is an example of a manifest file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
version="1.0.0.0"
processorArchitecture="X86"
name="CompanyName.ProductName.YourApp"
type="win32"
/>
<description>Your application description here.</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
</assembly>
```

Use the example above to create a manifest file for your application. If you place your manifest file in the same directory as your application, its controls will be rendered using the common controls version 6 theme engine. Your application now supports Windows XP themes.

For more information on Windows XP common controls, themes, and manifest files, consult Microsoft's online documentation.

Types of controls

Text Controls

Many applications use text controls to display text to the user. You can use:

- Edit controls, which allow the user to add text.
- Text viewing controls and labels, which do not allow user to add text.

Edit Controls

Edit controls display text to the user and allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
TEdit	Edit a single line of text.
TMemo	Edit multiple lines of text.
TMaskEdit	Adhere to a particular format, such as a postal code or phone number.
TRichEdit	Edit multiple lines of text using rich text format (VCL only).

TEdit and *TMaskEdit* are simple edit controls that include a single line text edit box in which you can type information. When the edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to *True*. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

TMaskEdit is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

TMemo and TRichEditcontrols allow the user to add several lines of text.

Edit controls have some of the following important properties:

Edit control properties

Property	Description
Text	Determines the text that appears in the edit box or memo control.
Font	Controls the attributes of text written in the edit box or memo control.
AutoSize	Enables the edit box to dynamically change its height depending on the currently selected font.
ReadOnly	Specifies whether the user is allowed to change the text.
MaxLength	Limits the number of characters in simple edit controls.
SelText	Contains the currently selected (highlighted) part of the text.
SelStart, SelLength	Indicate the position and length of the selected part of the text.

Memo and Rich Edit Controls

Both the *TMemo* and *TRichEdit* controls handle multiple lines of text.

TMemo is another type of edit box that handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

TRichEdit is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

Note: The rich edit control is available for VCL applications only.

In addition to the properties that all edit controls have, memo and rich edit controls include other properties, such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the Modified property.
- *Lines* contains the text as a list of strings.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names (VCL only).
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Text Viewing Controls

In CLX applications only, the text viewing controls display text but are read-only.

Use this component:	When you want users to do this:
TTextBrowser	Display a text file or simple HTML page that users can scroll through.
TTextViewer	Display a text file or simple HTML page. Users can scroll through the page or click links to view other pages and images.

TLCDNumber	Display numeric information in a digital display form.

TTextViewer acts as a simple viewer so that users can read and scroll through documents. With *TTextBrowser*, users can also click links to navigate to other documents and other parts of the same document. Documents visited are stored in a history list, which can be navigated using the *Backward, Forward*, and *Home* methods. *TTextViewer* and *TTextBrowser* are best used to display HTML-based text or to implement an HTML-based Help system.

TTextBrowser has the same properties as *TTextViewer* plus *Factory*. *Factory* determines the MIME factory object used to determine file types for embedded images. For example, you can associate filename extensions—such as .txt, .html, and .xml—with MIME types and have the factory load this data into the control.

Use the *FileName* property to add a text file, such as .html, to appear in the control at runtime.

To see an application using the text browser control, see ..\Delphi7\Demos\Clx\TextBrowser.

Labels

Labels display text and are usually placed next to other controls.

Use this component:	When you want users to do this:
TLabel	Display text on a nonwindowed control.
TStaticText	Display text on a windowed control.

You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is a non-windowed control (widget-based control in CLX applications), so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead.

Label properties include the following:

- *Caption* contains the text string for the label.
- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.
- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

Specialized Input Controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
TScrollBar	Select values on a continuous range
TTrackBar	Select values on a continuous range (more visually effective than a scroll bar)

TUpDown	Select a value from a spinner attached to an edit component (VCL applications only)
THotKey	Enter Ctrl/ Shift/ Alt keyboard sequences (VCL applications only)

Scroll Bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

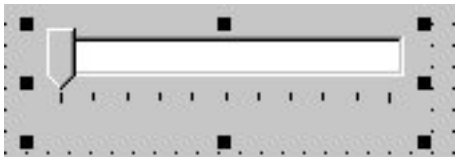
The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

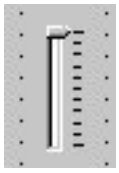
Track Bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.
- Use *SelEnd* and *SelStart* to highlight a selection range.
- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

Three views of the track bar component:





- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

Up-down Controls (VCL Only)

In VCL applications only, an up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

Hot Key Controls (VCL Only)

Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The HotKey property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

The hot key component can be assigned as the *ShortCut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.

Splitter Controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment to alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

Buttons and Similar Controls

Aside from menus, buttons provide the most common way to initiate an action or command in an application. Button-like controls include:

Use this component:	To do this:
TButton	Present command choices on buttons with text
TBitBtn	Present command choices on buttons with text and glyphs
TSpeedButton	Create grouped toolbar buttons
TCheckBox	Present on/off options
TRadioButton	Present a set of mutually exclusive choices
TToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions
TCoolBar	Display a collection of windowed controls within movable, resizable bands (VCL only)

Action lists let you centralize responses to user commands (actions) for objects such as menus and buttons that respond to those commands. See Using action lists for details on how to use action lists with buttons, toolbars, and menus.

You can custom draw buttons individually or application wide. See Developing the user interface.

Button Controls

Users click button controls to initiate actions. You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set Cancel to *True* if you want the button to trigger its OnClick event when the user presses Esc.
- Set Default to *True* if you want the Enter key to trigger the button's OnClick event.

Bitmap Buttons

A bitmap button (*TBitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph appears to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered on the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

Speed Buttons

Speed buttons (TSpeedButton), which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.
- If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

Check Boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to *True* to make the box appear checked by default.
- Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

Note: Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

Radio Buttons

Radio buttons, also called option buttons, present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See Grouping Controls for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value *True* or *False* to the Checked property to change the radio button's visual state.

Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *TToolBar* component, then right-click and choose New Button to add buttons to the toolbar.

The *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by using action lists or action bands.

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

Cool Bars (VCL Only)

A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows \System32 directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications.

- The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the **Object Inspector**, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands by adding new windowed controls from the palette.
- The *FixedOrder* property determines whether users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

List Controls

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
TListBox	A list of text strings
TCheckListBox	A list with a check box in front of each item
TComboBox	An edit box with a scrollable drop-down list
TTreeView	A hierarchical list
TListView	A list of (draggable) items with optional icons, columns, and headings
TDateTimePicker	A list box for entering dates or times (VCL applications only)

TMonthCalendar	A calendar for selecting dates (VCL applications only)

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see Working with string lists.

List Boxes and Check-list Boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select one or more choices from a list of possible options. The choices are represented using text, graphics, or both.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space (VCL only).
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding Graphics to Controls.

To create a simple list box

1. Within your project, drop a list box component from the **Tool palette** onto a form.
2. Size the list box and set its alignment as needed.
3. Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
4. Use the editor to enter free form text arranged in lines for the contents of the list box.
5. Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

Combo Boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

To create a combo box

1. Set the *Style* property to select the type of combo box you need:

- Use *csDropDown* to create an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list).
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding Graphics to Controls.

- Use *csSimple* to create a combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are displayed (VCL only).

2. Set the *DropDownCount* property to change the number of items displayed in the list.

At runtime, CLX combo boxes work differently than VCL combo boxes. With the CLX combo box, you can add an item to a drop-down list by entering text and pressing *Enter* in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

Tree Views

A tree view *(TTreeView*) displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of "+" and "–" buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships (VCL only).
- *ShowRoot* determines whether lines connecting the top-level items are displayed (VCL only).

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNodes*. *TTreeNodes* has methods for adding, deleting, and navigating the items in the tree view.

Tree views can display columns and subitems similar to list views in vsReport mode.

List Views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only).
- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

Date-time Pickers and Month Calendars

In CLX applications, the DateTimePicker component displays a list box for entering dates or times, while the MonthCalendar component presents a calendar for entering dates or ranges of dates. To use these components, you must have version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows \System32 directory) at both design time and runtime. They are not available for use in cross-platform applications.

Grouping Controls

A graphical interface is easier to use when related controls and information are presented in groups. Components for grouping components include:

Use this component:	When you want this:
TGroupBox	A standard group box with a title
TRadioGroup	A simple group of radio buttons
TPanel	A more visually flexible group of controls
TScrollBox	A scrollable region containing controls
TTabControl	A set of mutually exclusive notebook-style tabs
TPageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
THeaderControl	Resizable column headers

Group Boxes and Radio Groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the **Tool palette** and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the **Object Inspector**; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

Scroll Boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

Tab Controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the **Object Inspector**; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

Page Controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
[Delphi]
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

```
[C++]
TTabSheet *pTabSheet = new TTabSheet(PageControl1);
pTabSheet->PageControl = PageControl1;
```

To access the active page, use the *ActivePage* property. To change the active page, you can set either the *ActivePage* or the *ActivePageIndex* property.

Header Controls

A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

Display Controls

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime to identify the object.

Use this component or property:	To do this:
TStatusBar	Display a status region (usually at the bottom of a window)
TProgressBar	Show the amount of work completed for a particular task

Hint and ShowHint	Activate fly-by or "tooltip" Help
HelpContext and HelpFile	Link context-sensitive online Help

Status Bars

Although you can use a panel to make a status bar, it is simpler to use the *TStatusBar* component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the **Object Inspector**, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

Progress Bars

When your application performs a time-consuming operation, you can use a progress bar (*TProgressBar*) to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.

The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

Help and Hint Properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

Grids

Grids display information in rows and columns. If you're writing a database application, use the TDBCtrlGrid or TDBCtrlGrid component. Otherwise, use a standard draw grid or string grid.

Draw Grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.
- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.

- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

String Grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

Value List Editors (VCL Only)

TValueListEditor is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name using the *Values* property. *TValueListEditor* is not available for cross-platform programming.

The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using the *DisplayOptions* property (which also controls resize when you resize the control.)

You can control whether users can edit the Name column using the *KeyOptions* property. *KeyOptions* contains separate options to allow editing, adding new names, deleting names, and controlling whether new names must be unique.

You can control how users edit the entries in the Value column using the *ItemProps* property. Each item has a separate *TItemProp* object that lets you

- Supply an edit mask to limit the valid input.
- Specify a maximum length for values.
- Mark the value as read-only.
- Specify that the value list editor displays a drop-down arrow that opens a pick list of values from which the user can choose or an ellipsis button that triggers an event you can use for displaying a dialog in which users enter values.

If you specify that there is a drop-down arrow, you must supply the list of values from which the user chooses. These can be a static list (the *PickList* property of the *TItemProp* object) or they can be dynamically added at runtime using the value list editor's *OnGetPickList* event. You can also combine these approaches and have a static list that the *OnGetPickList* event handler modifies.

If you specify that there is an ellipsis button, you must supply the response that occurs when the user clicks that button (including the setting of a value, if appropriate). You provide this response by writing an *OnEditButtonClick* event handler.

Graphic Controls

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
TImage	Graphics files

TShape	Geometric shapes
TBevel	3-D lines and frames
TPaintBox	Graphics drawn by your program at runtime
TAnimate	AVI files (VCL applications only); GIF files (CLX applications only)

Notice that these include common paint routines (*Repaint, Invalidate*, and so on) that never need to receive focus.

To create a graphic control, see Creating a graphic control.

Images

The image component (*TImage*) displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center, AutoSize, Stretch*, and *Transparent* to set display options. For more information, see Overview of Graphics Programming.

Shapes

The shape component displays a geometric shape. It is a nonwindowed control (a widget-based control in CLX applications) and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

Paint Boxes

The paint box (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see Overview of Graphics Programming.

Animation Control

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip (VCL applications) or a GIF clip (CLX applications). An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE).

Following are some of the properties of an animation component:

- *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After set ting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.
- Set *AutoSize* to *True* to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Set *CommonAVI* to display one of the common Windows AVI clips provided in Shell32.DLL.

- Specify when to start and interrupt the animation by setting the *Active* property to *True* and *False*, respectively, and how many repetitions to play by setting the *Repetitions* property.
- The *Timers* property lets you display the frames using a timer. This is useful for synchronizing the animation sequence with other actions, such as playing a sound track.

Working with graphics and multimedia

Working with Graphics and Multimedia: Overview

Graphics and multimedia elements can add polish to your applications. You can introduce these features into your application in a variety of ways. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, you can use special components that can play audio and video clips.

This following topics describe how to enhance your applications by introducing graphics or multimedia elements:

- Overview of Graphics Programming
- Working with multimedia

Overview of Graphics Programming

In VCL applications, the graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications. CLX graphics components defined in the QGraphics unit encapsulate the Qt graphics widgets for adding graphics to cross-platform applications.

To draw graphics in an application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it manages the device context for you, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles (drawings in CLX applications). Canvases are available only at runtime, so you do all your work with canvases by writing code.

Note: Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI functions on the canvas. The *Handle* property of the canvas is the device context Handle.

In CLX applications, *TCanvas* is a wrapper resource manager around a Qt painter. The *Handle* property of the canvas is a typed pointer to an instance of a Qt painter object. Having this instance pointer exposed allows you to use low-level Qt graphics library functions that require an instance pointer to a painter object QPainterH.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* message (VCL applications) or event (CLX applications).

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The following topics describe how to use graphics components to simplify your coding.

- Refreshing the screen
- Types of graphic objects
- Common properties and methods of canvases
- Handling multiple drawing objects in an application
- Drawing on a bitmap
- Loading and saving graphics files
- Using the Clipboard with Graphics
- Rubber banding example

Refreshing the Screen

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages on Windows, which the VCL routes to *OnPaint* events. (In CLX applications, a paint event is generated, and routed to *OnPaint* events.) If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if using the VCL, you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated, Windows does not. In the Windows operating system anything drawn on the screen is permanent. When a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. For more information about the WM_PAINT message, see the Windows online Help.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that *TImage* displays. You can also set the *Proportional* property to ensure that the image can be fully displayed in the image control without any distortion. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the screen device (VCL applications) or the painter (CLX applications), so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

Types of Graphic Objects

The component library provides the following graphic objects. These objects have methods to draw on the canvas, which are described in Using Canvas methods to draw graphic objects and to load and save to graphics files, as described in Loading and saving graphics files

Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image.
Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file (::ICO file).
Metafile (VCL applications only) Drawing (CLX applications only)	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles or drawings are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles and drawings do not display as fast as bitmaps. Use a metafile or drawing when versatility or precision is more important than performance.

Common Properties and Methods of Canvas

The following table lists the commonly used properties of the Canvas object.

Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in Using the properties of the Canvas object.

Here is a list of several methods you can use:

Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FloodFill (VCL only)	Fills an area of the canvas using the current brush.

FrameRect (VCL only)	Draws a rectangle using the Brush of the canvas to draw the border.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.

These methods are described in more detail in Using Canvas methods to draw graphic objects.

Using the Properties of the Canvas Object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This topic describes:

- Using pens.
- Using brushes.
- Reading and setting pixels.

Using Pens

The Pen property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change:

- Color property changes the pen color.
- Width property changes the pen width.
- Style property changes the pen style.
- Mode property changes the pen mode.

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

You can use TPenRecall for quick saving off and restoring the properties of pens.

Changing the Pen Color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
[Delphi]
procedure TForm1.PenColorClick(Sender: TObject);
begin
Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

```
[C++]
void __fastcall TForm1::PenColorClick(TObject *Sender)
{
  Canvas->Pen->Color = PenColor->ForegroundColor;
}
```

Changing the Pen Width

A pen's width determines the thickness, in pixels, of the lines it draws.

Note: When the thickness is greater than 1, Windows always draws solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's Width property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
[Delphi]
procedure TForm1.PenWidthChange(Sender: TObject);
begin
Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

```
[C++]
void __fastcall TForm1::PenWidthChange(TObject *Sender)
{
  Canvas->Pen->Width = PenWidth->Position;         // set the pen width directly
  PenSize->Caption = IntToStr(PenWidth->Position); // convert to string
}
```

Changing the Pen Style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

Note: For CLX applications deployed under Windows, Windows does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1. Select all six pen-style buttons and select the **Object Inspector** ▸ **Events** ▸ **OnClick event** and in the Handler column, type SetPenStyle.

   The Code editor generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

2. Fill in the click-event handler by setting the pen's style depending on the value of Sender, which is the control that sent the click event:

```
[Delphi]
procedure TForm1.SetPenStyle(Sender: TObject);
begin
with Canvas.Pen do
begin
if Sender = SolidPen then Style := psSolid
else if Sender = DashPen then Style := psDash
else if Sender = DotPen then Style := psDot
else if Sender = DashDotPen then Style := psDashDot
else if Sender = DashDotDotPen then Style := psDashDotDot
else if Sender = ClearPen then Style := psClear;
end;
end;
```

```
[C++]
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender == SolidPen)
    Canvas->Pen->Style = psSolid;
  else if (Sender == DashPen)
    Canvas->Pen->Style = psDash;
  else if (Sender == DotPen)
    Canvas->Pen->Style = psDot;
  else if (Sender == DashDotPen)
    Canvas->Pen->Style = psDashDot;
  else if (Sender == DashDotDotPen)
```

```
    Canvas->Pen->Style = psDashDotDot;
'  else if (Sender == ClearPen)
    Canvas->Pen->Style = psClear;
}
```

```
[C++]
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Pen->Style = (TPenStyle) ((TSpeedButton *)Sender)->Tag;
}
```

Changing the Pen Mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on.

Getting the Pen Position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its PenPos property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the MoveTo method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
[Delphi]
Canvas.MoveTo(0, 0);
```

```
[C++]
Canvas->MoveTo(0, 0);
```

Note: Drawing a line with the LineTo method also moves the current position to the endpoint of the line.

Using Brushes

The Brush property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property changes the fill color.
- Style property changes the brush style.
- Bitmap property uses a bitmap as a brush pattern.

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

You can use TBrushRecall for quick saving off and restoring the properties of brushes.

Changing the Brush Color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar :

```
[Delphi]
procedure TForm1.BrushColorClick(Sender: TObject);
begin
Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

```
[C++]
void __fastcall TForm1::BrushColorClick(TObject *Sender)
{
  Canvas->Brush->Color = BrushColor->BackgroundColor;
}
```

Changing the Brush Style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its Style property to one of the predefined values: bsBDiagonal, bsClear, bsCross, bsDiagCross, bsFDiagonal, bsHorizontal, bsSolid, or bsVertical. Cross-platform applications include the predefined values of bsDense1 through bsDense7.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons are selected, the **Object Inspector** ▸ **Events** ▸ **OnClick** is set, and the OnClick handler is named SetBrushStyle.

Here is the handler code:

```
[Delphi]
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

```
[C++]
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
  if (Sender == SolidBrush)
    Canvas->Brush->Style = bsSolid;
```

```
  else if (Sender == ClearBrush)
    Canvas->Brush->Style = bsClear;
  else if (Sender == HorizontalBrush)
    Canvas->Brush->Style = bsHorizontal;
  else if (Sender == VerticalBrush)
    Canvas->Brush->Style = bsVertical;
  else if (Sender == FDiagonalBrush)
    Canvas->Brush->Style = bsFDiagonal;
  else if (Sender == BDiagonalBrush)
    Canvas->Brush->Style = bsBDiagonal;
  else if (Sender == CrossBrush)
    Canvas->Brush->Style = bsCross;
  else if (Sender == DiagCrossBrush)
    Canvas->Brush->Style = bsDiagCross;
}
```

```
[C++]
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
   if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Brush->Style = (TBrushStyle) ((TSpeedButton *)Sender)->Tag;
}
```

Setting the Brush Bitmap Property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```
[Delphi]
var
Bitmap: TBitmap;
begin
Bitmap := TBitmap.Create;
try
Bitmap.LoadFromFile('MyBitmap.bmp');
Form1.Canvas.Brush.Bitmap := Bitmap;
Form1.Canvas.FillRect(Rect(0,0,100,100));
finally
Form1.Canvas.Brush.Bitmap := nil;
Bitmap.Free;
end;
end;
```

```
[C++]
BrushBmp->LoadFromFile("MyBitmap.bmp");
Form1->Canvas->Brush->Bitmap = BrushBmp;
Form1->Canvas->FillRect(Rect(0,0,100,100));
```

Note: The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

Reading and Setting Pixels

You will notice that every canvas has an indexed Pixels property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

Note: Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the TBitmap.ScanLine property

Using Canvas Methods to Draw Graphic Objects

This topic shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines.
- Drawing shapes.
- Drawing rounded rectangles.
- Drawing polygons.

Drawing Lines and Polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

Drawing Lines

To draw a straight line on a canvas, use the LineTo method of the canvas.

LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
[Delphi]
procedure TForm1.FormPaint(Sender: TObject);
begin
with Canvas do
begin
MoveTo(0, 0);
LineTo(ClientWidth, ClientHeight);
MoveTo(0, ClientHeight);
LineTo(ClientWidth, 0);
end;
end;
```

```
[C++]
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->MoveTo(0,0);
  Canvas->LineTo(ClientWidth, ClientHeight);
  Canvas->MoveTo(0, ClientHeight);
  Canvas->LineTo(ClientWidth, 0);
}
```

Drawing Polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the Polyline method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the MoveTo method and the LineTo method because it eliminates a lot of call overhead.

Drawing Shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This topic describes:

- Drawing rectangles and ellipses.
- Drawing rounded rectangles.
- Drawing polygons.

Drawing Rectangles and Ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's Rectangle method or Ellipse method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

Drawing Rounded Rectangles

To draw a rounded rectangle on a canvas, call the canvas's RoundRect method.

The first four parameters passed to RoundRect are a bounding rectangle, just as for the Rectangle method or the Ellipse method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

Drawing Polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

Handling Multiple Drawing Objects in Your Application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use.
- Change the tool with speed buttons.
- Use drawing tools.

Keeping Track of Which Drawing Tool to Use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time.

You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won"t be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Delphi provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Delphi's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
[Delphi]
type
TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

```
[C++]
typedef enum {dtLine, dtRectangle, dtEllipse, dtRoundRect} TDrawingTool;
```

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for "drawing tool").

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
[Delphi]
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use the Delphi language's type-checking to prevent many errors. A variable of type *TDrawingTool* can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

```
[Delphi]
type
```

```
TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
TForm1 = class(TForm)
...{ method declarations }
public
Drawing: Boolean;
Origin, MovePt: TPoint;
DrawingTool: TDrawingTool;{ field to hold current tool }
end;
```

```
[C++]
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:
public:
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
  TDrawingTool DrawingTool;  // field to hold current tool
};
```

Changing the Tool with Speed Buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```
[Delphi]
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
DrawingTool := dtLine;
end;
procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
DrawingTool := dtRectangle;
end;
procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
DrawingTool := dtEllipse;
end;
procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
DrawingTool := dtRoundRect;
end;
```

```
[C++]
void __fastcall TForm1::LineButtonClick(TObject *Sender)  // LineButton
{
  DrawingTool = dtLine;
}
```

```
void __fastcall TForm1::RectangleButtonClick(TObject *Sender) // RectangleButton
{
  DrawingTool = dtRectangle;
}
void __fastcall TForm1::EllipseButtonClick(TObject *Sender)  // EllipseButton
{
  DrawingTool = dtEllipse;
}
void __fastcall TForm1::RoundedRectButtonClick(TObject *Sender) // RoundRectBtn
{
  DrawingTool = dtRoundRect;
}
```

Using Drawing Tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This topic describes:

- Drawing shapes.
- Sharing code among event handlers.

Drawing Shapes

Drawing shapes is just as easy as drawing lines. Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

```
[Delphi]
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                             X,Y: Integer);
begin
case DrawingTool of
dtLine:
begin
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(X, Y)
end;
dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
end;
Drawing := False;
end;
```

```
[C++]
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
                                    TShiftState Shift, int X, int Y){
  switch (DrawingTool)
```

```
  {
    case dtLine:
      Canvas->MoveTo(Origin.x, Origin.y);
      Canvas->LineTo(X, Y);
      break;
    case dtRectangle:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y);
      break;
    case dtEllipse:
      Canvas->Ellipse(Origin.x, Origin.y, X, Y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y, (Origin.x - X)/2,
                        (Origin.y - Y)/2);
      break;
  }
  Drawing = false;
}
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
[Delphi]
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.Pen.Mode := pmNotXor;
case DrawingTool of
dtLine: begin
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(MovePt.X, MovePt.Y);
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(X, Y);
end;
dtRectangle: begin
Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
end;
dtEllipse: begin
Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
end;
dtRoundRect: begin
Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
end;
end;
MovePt := Point(X, Y);
end;
Canvas.Pen.Mode := pmCopy;
end;
```

```
[C++]
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
```

```
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    switch (DrawingTool)
    {
      case dtLine:
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(MovePt.x, MovePt.y);
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(X, Y);
        break;
      case dtRectangle:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Rectangle(Origin.x, Origin.y, X, Y);
        break;
      case dtEllipse:
        Canvas->Ellipse(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Ellipse(Origin.x, Origin.y, X, Y);
        break;
      case dtRoundRect:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y,
                          (Origin.x - MovePt.x)/2,(Origin.y - MovePt.y)/2);
        Canvas->Rectangle(Origin.x, Origin.y, X, Y,
                          (Origin.x - X)/2, (Origin.y - Y)/2);
        break;
    }
    MovePt = Point(X, Y);
  }
  Canvas->Pen->Mode = pmCopy;
}
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next topic shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

Sharing Code Among Event Handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form:

1. Add the method declaration to the form object.

   You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.

2. Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

Drawing On a Graphic

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

Note: Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

Making Scrollable Graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

Adding an Image Control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Note: Adding Graphics to Controls shows how to use graphics in controls.

Placing the Control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

Setting the Initial Bitmap Size

When you place an image control, it is simply a container. However, you can set the image control's Picture property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in Loading And Saving Graphics Files.

To create a blank bitmap when the application starts

1. Attach a handler to the *OnCreate* event for the form that contains the image.
2. Create a bitmap object, and assign it to the image control's *Picture.Graphic* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1's OnCreate* event:

```
[Delphi]

procedure TForm1.FormCreate(Sender: TObject);
var
Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
Bitmap := TBitmap.Create;{ construct the bitmap object }
  Bitmap.Width := 200;{ assign the initial width... }
  Bitmap.Height := 200;{ ...and the initial height }
  Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
  Bitmap.Free; {We are done with the bitmap, so free it }
end;
```

```
[C++]
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap = new Graphics::TBitmap();   // create the bitmap object
  Bitmap->Width = 200;               // assign the initial width...
  Bitmap->Height = 200;              // ...and the initial height
  Image->Picture->Graphic = Bitmap;  // assign the bitmap to the image control
  delete Bitmap;                     // free the bitmap object
}
```

Assigning the bitmap to the picture's *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

Drawing On the Bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
x,y : integer;
Bitmap : TBitmap;
P : PByteArray;
begin
Bitmap := TBitmap.create;
try
Bitmap.LoadFromFile("C:\Program Files\Borland\Delphi 4\Images\Splash\256color
\factory.bmp");
for y := 0 to Bitmap.height -1 do
begin
P := Bitmap.ScanLine[y];
```

```
for x := 0 to Bitmap.width -1 do
P[x] := y;
end;
canvas.draw(0,0,Bitmap);
finally
Bitmap.free;
end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Graphics::TBitmap *pBitmap = new Graphics::TBitmap();
// This example shows drawing directly to the Bitmap
Byte *ptr;
try
{
pBitmap->LoadFromFile("C:\\Program Files\\Borland\\CBuilder\\Images\\Splash\\256color\
\factory.bmp ");
for (int y = 0; y < pBitmap->Height; y++)
{
ptr = pBitmap->ScanLine[y];
for (int x = 0; x < pBitmap->Width; x++)
ptr[x] = (Byte)y;
}
Canvas->Draw(0,0,pBitmap);
}
catch (...)
{
ShowMessage("Could not load or alter bitmap");
}
delete pBitmap;
}
```

Note: For CLX applications, change Windows- and VCL-specific code so that your application can run on Linux. For example, the pathnames in Linux use a forward slash / as a delimiter.

Loading and Saving Graphics Files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, (pngs and xpms in CLX applications) and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in these topics:

- Loading a picture from a file.
- Saving a picture to a file.
- Replacing the picture.

Loading a Picture from a File

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

```
[Delphi]
procedure TForm1.Open1Click(Sender: TObject);
begin
if OpenPictureDialog1.Execute then
begin
CurrentFile := OpenPictureDialog1.FileName;
Image.Picture.LoadFromFile(CurrentFile);
end;
end;
```

```
[C++]
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenPictureDialog1->Execute())
  {
    CurrentFile = OpenPictureDialog1->FileName;
    Image->Picture->LoadFromFile(CurrentFile);
  }
}
```

Saving a Picture to a File

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next topic.

The following pair of event handlers, attached to the **File** ▸ **Save** and **File** ▸ **Save As** menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```
[Delphi]
procedure TForm1.Save1Click(Sender: TObject);
begin
if CurrentFile <> '' then
Image.Picture.SaveToFile(CurrentFile){ save if already named }
else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
if SaveDialog1.Execute then{ get a file name }
begin
CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
Save1Click(Sender);{ then save normally }
```

```
end;
end;
```

```
[C++]
void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (!CurrentFile.IsEmpty())
    Image->Picture->SaveToFile(CurrentFile);   // save if already named
else SaveAs1Click(Sender);                     // otherwise get a name
}
void __fastcall TForm1::SaveAs1Click(TObject *Sender)
{
  if (SaveDialog1->Execute())              // get a file name
  {
    CurrentFile = SaveDialog1->FileName;  // save user-specified name
    Save1Click(Sender);                   // then save normally
  }
}
```

Replacing the Picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic , but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box.

With such a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the **File** ▸ **New** menu item's OnClick event. Here's an example:

```
[Delphi]
procedure TForm1.New1Click(Sender: TObject);
var
Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
with NewBMPForm do
begin
ActiveControl := WidthEdit;{ make sure focus is on width field }
WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
if ShowModal <> idCancel then{ continue if user doesn"t cancel dialog box }
begin
Bitmap := TBitmap.Create;{ create fresh bitmap object }
Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
CurrentFile := "";{ indicate unnamed file }
     Bitmap.Free;
end;
end;
end;
```

```
[C++]
void __fastcall TForm1::New1Click(TObject *Sender)
```

```
{
Graphics::TBitmap *Bitmap;
  // make sure focus is on width field
 NewBMPForm->ActiveControl = NewBMPForm->WidthEdit;
  // initialize to current dimensions as default ...
     NewBMPForm->WidthEdit->Text = IntToStr(Image->Picture->Graphic->Width);
NewBMPForm->HeightEdit->Text = IntToStr(Image->Picture->Graphic->Height);
  if (NewBMPForm->ShowModal() != IDCANCEL){        // if user does not cancel dialog...
Bitmap = new Graphics::TBitmap();             // create a new bitmap object
    // use specified dimensions
     Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
Image->Picture->Graphic = Bitmap;            // replace graphic with new bitmap
CurrentFile = EmptyStr;                      //indicate unnamed file
        delete Bitmap;
}
}
```

Note: Assigning a new bitmap to the picture object's Graphic property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

Using the Clipboard with Graphics

You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the clipboard object in your application, you must add the Clipbrd (QClipbrd in CLX applications) unit to the uses clause of any unit that needs to access clipboard data.

For CLX applications, data that is stored on the clipboard is stored as a MIME type with an associated *TStream* object. CLX applications provide predefined constants for the following MIME types.

CLX MIME types and constants

MIME type	CLX constant
'image/delphi.bitmap'	SDelphiBitmap
'image/delphi.component'	SDelphiComponent
'image/delphi.picture'	SDelphiPicture
'image/delphi.drawing'	SDelphiDrawing

Copying Graphics to the Clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an **Edit** ▸ **Copy** menu item:

```
[Delphi]
procedure TForm1.Copy1Click(Sender: TObject);
begin
```

```
  Clipboard.Assign(Image.Picture)
end.
```

```
[C++]
void __fastcall TForm1::Copy1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

Cutting Graphics to the Clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the **Edit** ▸ **Cut** menu item:

```
[Delphi]
procedure TForm1.Cut1Click(Sender: TObject);
var
ARect: TRect;
begin
Copy1Click(Sender);{ copy picture to clipboard }
with Image.Canvas do
begin
CopyMode := cmWhiteness;{ copy everything as white }
ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
CopyMode := cmSrcCopy;{ restore normal mode }
end;
end;
```

```
[C++]
void __fastcall TForm1::Cut1Click(TObject *Sender)
{
  TRect ARect;
  Copy1Click(Sender);          // copy picture to clipboard
  Image->Canvas->CopyMode = cmWhiteness; // copy everything as white
  ARect = Rect(0, 0, Image->Width, Image->Height); // get dimensions of image
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect); // copy bitmap over self
  Image->Canvas->CopyMode = cmSrcCopy; // restore default mode
}
```

Pasting Graphics from the Clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard:

1 Call the clipboard's HasFormat method (VCL applications) to see whether the clipboard contains a graphic.

HasFormat (or *Provides* in CLX applications) is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter. To test for graphics on the Windows platform, you pass *CF_BITMAP*. In CLX applications, you pass *SDelphiBitmap*.

2 Assign the clipboard to the destination.

Note: The following VCL code shows how to paste a picture from the clipboard into an image control in response to a click on an **Edit** ▸ **Paste** menu item:

```
[Delphi]
procedure TForm1.PasteButtonClick(Sender: TObject);
var
Bitmap: TBitmap;
begin
if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows clipboard? )
begin
Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

```
[C++]
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
Graphics::TBitmap *Bitmap;
if (Clipboard()->HasFormat(CF_BITMAP)){
Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

Note: The same example in a CLX application would look as follows:

```
[Delphi]
procedure TForm1.PasteButtonClick(Sender: TObject);
var
Bitmap: TBitmap;
begin
if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? )
begin
Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

```
[C++]
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
QGraphics::TBitmap *Bitmap;
if (Clipboard()->Provides(SDelphiBitmap)){
Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

The graphic on the clipboard could come from this application, or it could have been copied from another application, such as Microsoft Paint. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

Rubber Banding Example

This example describes the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code covered in this topic is taken from a sample application located in the Demos\Doc\Graphexdirectory. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse.
- Adding a field to a form object to track mouse actions.
- Refining line drawing.

Responding to the Mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This topic describes:

- What's in a mouse event.
- Responding to a mouse-down action.
- Responding to a mouse-up action.
- Responding to a mouse move.

What's in a Mouse Event

A mouse event occurs when a user moves the mouse in the user interface of an application. The VCL has three mouse events.

Mouse events

Event	Description
OnMouseDown event	Occurs when the user presses a mouse button with the mouse pointer over a control.
OnMouseMove event	Occurs when the user moves the mouse while the mouse pointer is over a control.
OnMouseUp event	Occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

When an application detects a mouse action, it calls whatever event handler you've defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Mouse-event parameters

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which mouse button was involved: *mbLeft, mbMiddle*, or *mbRight*

Shift	Indicates the state of the Alt, Ctrl, and Shift keys at the time of the mouse action
X, Y	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

Note: Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record *mbLeft* as the value of the *Button* parameter.

Responding to a Mouse-down Action

Whenever the user presses a button on the mouse, an OnMouseDown event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The Code editor generates an empty handler for a mouse-down event on the form:

```
[Delphi]
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
end;
```

```
[C++]
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
}
```

Responding to a Mouse-up Action

An OnMouseUp event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

Responding to a Mouse Move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
[Delphi]
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

```
[C++]
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);
}
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

Adding a Field to a Form Object to Track Mouse Actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

Refining Line Drawing

With fields in place to track various points, you can refine an application's line drawing.

Tracking the Origin Point

When drawing lines, track the point where the line starts with the *Origin* field. *Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```
[Delphi]
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Drawing := True;
Canvas.MoveTo(X, Y);
Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
```

```
Shift: TShiftState; X, Y: Integer);
begin
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
Canvas.LineTo(X, Y);
Drawing := False;
end;
```

```
[C++]
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
  Canvas->LineTo(X, Y);                // draw line from PenPos to (X, Y)
  Drawing = false;                     // clear the Drawing flag
}
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions—the application does not yet support "rubber banding."

Tracking Movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position*, not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```
[Delphi]
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
Canvas.LineTo(X, Y);
end;
end;
```

```
[C++]
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(X, Y);
  }
}
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

MovePt must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
[Delphi]
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Drawing := True;
Canvas.MoveTo(X, Y);
Origin := Point(X, Y);
MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
Canvas.LineTo(X, Y);{ draw the new line }
end;
MovePt := Point(X, Y);{ record point for next move }
Canvas.Pen.Mode := pmCopy;
end;
```

```
[C++]
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
  MovePt = Point(X, Y);      // record last endpoint
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(MovePt.x, MovePt.y);  // erase old line
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point again
    Canvas->LineTo(X, Y);                // draw new line
  }
  MovePt = Point(X, Y);      // record new endpoint
  Canvas->Pen->Mode = pmCopy;
}
```

Now you get a "rubber band" effect when you draw the line. By changing the pen's mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you're actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

Working with Multimedia

You can add multimedia components to your applications. To do this, you can use either the TAnimate component on the Win32 (Common Controls in CLX applications) page or the TMediaPlayer component (not available in CLX applications) on the System category of the **Tool palette**. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.

This topic discusses:

- Adding silent video clips to an application
- Adding audio and/or video clips to an application

Adding Silent Video Clips to an Application

With the animation control, you can add silent video clips to your application:

To add silent videop clips

1. Double-click the *TAnimate* icon on the Win32 (Common Control in CLX applications) catgory of the **Tool palette**. This automatically puts an animation control on the form window in which you want to display the video clip.
2. Using the **Object Inspector**, select the Name property and enter a new *nam*e for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers).

   Always work directly with the **Object Inspector** when setting design time properties and creating event handlers.
3. Do one of the following:

   - Select the CommonAVI property and choose one of the AVIs available from the drop-down list; or
   - Select the resource of an AVI using the ResName or ResID properties. Use ResHandle to indicate the module that contains the resource identified by *ResName* or *ResID*; or
   - Select the FileName property and click the ellipsis (...) button, choose an AVI file (GIF in CLX applications) from any available local or network directories and click Open in the Open AVI or Open GIF dialog (Windows and cross-platform applications).

   This loads the AVI or GIF file into memory. If you want to display the first frame of the AVI or GIF clip on-screen until it is played using the *Active* property or the *Play* method, then set the Open property to *True*.
4. Set the Repetitions property to the number of times you want to the AVI or GIF clip to play. If this value is 0, then the sequence is repeated until the Stop method is called.
5. Make any other changes to the animation control settings. For example, *if* you want to change the first frame displayed when animation control opens, then set the StartFrameproperty to the desired frame value.
6. Set the Active property to *True* using the drop-down list or write an event handler to run the AVI or GIF clip when a specific event takes place at runtime. For example, to activate the AVI or GIF clip when a button object is clicked, write the button's OnClick event specifying that. You may also call the Play method to specify when to play the AVI (VCL only).

Note: If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

For more information on using the animation control, see the topic called Example of adding silent video clips.

Example of Adding Silent Video Clips

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the Unit1.pas file as Frmlogo.pas and save the Project1.dpr file as Logo.dpr. Then:

1. Double-click the animate icon from the Win32 category of the **Tool palette**.
2. Using the **Object Inspector**, set its Name property to *Logo1*.
3. Select its FileName property, click the ellipsis (...) button, locate and choose an AVI file. Then click Open in the Open AVI dialog.

   This loads the AVI file into memory.
4. Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
5. Set its Repetitions property to 5.
6. Click the form to bring focus to it and set its Name property to *LogoForm1* and its Caption property to *Logo Window*. Now decrease the height of the form to right-center the animation control on it.
7. Double-click the form's *OnActivate* event and write the following code to run the AVI clip when the form is in focus at runtime:

```
[Delphi]
Logo1.Active := True;
```

```
[C++]
Logo1->Active = true;
```

8. Double-click the Label icon on the Standard category of the **Tool palette**. Select its Caption property and enter *Welcome to Cool Images 4.0*. Now select its Font property, click the ellipsis (...) button and choose Font Style: Bold, Size: 18, Color: Navy from the Font dialog and click OK. Click and drag the label control to center it on the form.
9. Click the animation control to bring focus back to it. Double-click its *OnStop* event and write the following code to close the form when the AVI file stops:

```
[Delphi]
LogoForm1.Close;
```

```
[C++]
LogoForm1->Close();
```

Select **Run** ▸ **Run** to execute the animated logo window.

Adding Audio and/or Video Clips to an Application

With the media player component, you can add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

Note: Audio support is not available in cross-platform applications.

To add an audio and/or video clip to an application:

1. Double-click the media player icon on the System category of the **Tool palette**. This automatically put a media player control on the form window in which you want the media feature.
2. Using the **Object Inspector**, select the Name*property* and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.)

   Always work directly with the **Object Inspector** when setting design time properties and creating event handlers.
3. Select the DeviceType property and choose the appropriate device type to open using the *AutoOpen* property or the *Open* method. (If *DeviceType* is dtAutoSelect the device type is selected based on the file extension of the media file specified by the *FileName* property.) For more information on device types and their functions, see the table below.
4. If the device stores its media in a file, specify the name of the media file using the *FileName* property. Select the FileName property, click the ellipsis (...) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.
5. Set the AutoOpen property to *True*. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If *AutoOpen* is *False*, the device must be opened with a call to the Open method.
6. Set the AutoEnable property to *True* to automatically enable or disable the media player buttons as required at runtime; or, double-click the EnabledButtons property to set each button to *True* or *False* depending on which ones you want to enable or disable.

   The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).
7. Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the Align property and choosing the appropriate align position from the drop down list.

   If you want the media player to be invisible at runtime, set the Visible property to *False* and control the device by calling the appropriate methods (Play, Pause, Stop, Next, Previous, Step, Back, StartRecording, Eject).
8. Make any other changes to the TMediaPlayer control settings. For example, if the media requires a display window, set the Display property to the control that displays the media. If the device uses multiple tracks, set the Tracks property to the desired track.

Multimedia device types and their functions

Device Type	Software/Hardware used	Plays	Uses Tracks	Uses a Display Window
dtAVIVideo	AVI Video Player for Windows	AVI Video files	No	Yes
dtCDAudio	CD Audio Player for Windows or a CD Audio Player	CD Audio Disks	Yes	No
dtDAT	Digital Audio Tape Player	Digital Audio Tapes	Yes	No
dtDigitalVideo	Digital Video Player for Windows	AVI, MPG, MOV files	No	Yes
dtMMMovie	MM Movie Player	MM film	No	Yes
dtOverlay	Overlay device	Analog Video	No	Yes
dtScanner	Image Scanner	N/A for Play (scans images on Record)	No	No

dtSequencer	MIDI Sequencer for Windows	MIDI files	Yes	No
dtVCR	Video Cassette Recorder	Video Cassettes	No	Yes
dtWaveAudio	Wave Audio Player for Windows	WAV files	No	No

For more information on using the media player control, see the topic called Example of Adding Audio and/or Video Clips.

Example of Adding Audio and/or Video Clips (VCL Only)

This example runs an AVI video clip of a multimedia advertisement.

To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

1. Double-click the media player icon on the System category of the **Tool palette**.
2. Using the **Object Inspector**, set the Name property of the media player to *VideoPlayer1*.
3. Select its DeviceType property and choose dtAVIVideo from the drop-down list.
4. Select its FileName property, click the ellipsis (...) button, locate and choose an AVI file. Click Open in the Open dialog.
5. Set its AutoOpen property to *True* and its Visible property to *False*.
6. Double-click the Animate icon from the Win32 category of the **Tool palette**. Set its AutoSize property to *False*, its Height property to 175 and Width property to 200. Click and drag the animation control to the top left corner of the form.
7. Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.
8. Click the form to bring focus to it and select its Name property and enter *Delphi_Ad*. Now resize the form to the size of the animation control.
9. Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
[Delphi]
VideoPlayer1.Play;
```

```
[C++]
VideoPlayer1->Play();
```

Choose **Run** ▶ **Run** to execute the AVI video.

Writing multi-threaded applications

Writing Multi-threaded Applications

Several objects make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by:

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

Note: Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example, Windows 9x only simulates multiprocessing, even if the underlying hardware supports it.

The following topics discuss support for threads:

- Defining Thread Objects
- Coordinating Threads
- Executing Thread Objects
- Debugging Multi-threaded Applications

Defining Thread Objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Note: Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the BeginThread function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in Coordinating Threads.

To use a thread object in your application

1. Create a new descendant of TThread, choose **File** ▸ **New** ▸ **Other** from the main menu.
2. In the New Items dialog box under Delphi Files, double-click Thread Object and enter a class name, such as *TMyThread*.
3. Check the Named Thread check box and enter a thread name (VCL applications only).
4. Click OK, the **Code Editor** creates a new unit file to implement the thread.

For more information on naming threads, see Naming a Thread.

Note: Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog box does not automatically prepend a 'T' to the front of the class name you provide.

The automatically generated unit file contains the skeleton code for your new thread class. If you named your thread *TMyThread*, it would look like the following:

```
[Delphi]
unit Unit2;
interface
uses
Classes;
type
TMyThread = class(TThread)
private
{ Private declarations }
protected
procedure Execute; override;
end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
{ Place thread code here }
end;
end.
```

```
[C++]
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit2.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
// ---- Place thread code here ----
```

```
}
//---------------------------------------------------------------------------
```

In the automatically generated unit file, you

- Optionally, initialize the thread.
- Write the thread function by filling in the *Execute* method.
- Optionally, write clean-up code.

Initializing the Thread

If you want to write initialization code for your new thread class, you must override the *Create* method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the Priority property.

If writing a Windows-only application, *Priority* values fall along a scale, as described in the following table:

Thread priorities

Value	Priority
tpIdle	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with *tpIdle* priority.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.
tpHighest	The thread's priority is two points above normal.
tpTimeCritical	The thread gets highest priority.

Warning: Boosting the thread priority of a CPU intensive operation may "starve" other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application's performance:

```
[Delphi]
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
Priority := tpIdle;
end;
```

```
[C++]
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
Priority = tpIdle;
}
//---------------------------------------------------------------------------
```

Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the FreeOnTerminate property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.

Writing the Thread Function

The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

When implementing the *Execute* method, you can manage these issues by:

- Using thread-local variables
- Avoiding simultaneous access
- Waiting for other threads
- Checking for termination by other threads
- Handling exceptions in the thread function

Using the Main VCL Thread

When you use objects from the class hierarchy, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside to access VCL objects. This is the thread that handles all Windows messages received by components in your application.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's Synchronize method. For example:

```
[Delphi]
procedure TMyThread.PushTheButton;
begin
Button1.Click;
end;
...
procedure TMyThread.Execute;
```

```
begin
...
Synchronize(PushTheButton);
...
end;
```

```
[C++]
void __fastcall TMyThread::PushTheButton(void)
{
Button1->Click();
}
void __fastcall TMyThread::Execute()
{
...
Synchronize((TThreadMethod)PushTheButton);
...
}
```

Synchronize waits for the main thread to enter the message loop and then executes the passed method.

Note: Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to VCL objects in console applications.

You do not always need to use the main thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the VCL thread to enter its message loop. You do not need to use the *Synchronize* method for the following objects:

Object	Description
Data access component	Data access components are thread-safe as follows: For BDE-enabled datasets, each thread must have its own database session component. The one exception to this is when you are using Microsoft Access drivers, which are built using a Microsoft library that is not thread-safe. For dbExpress, as long as the vendor client library is thread-safe, the dbExpress components will be thread-safe. ADO and InterBaseExpress components are thread-safe. When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset. For more information about using database sessions with threads in BDE-enabled applications, see Managing multiple sessions.
Control	Controls are not thread-safe.
Graphic	Graphics objects are thread-safe. You do not need to use the main VCL thread to access TFont, TPen, TBrush, TBitmap, TMetafile (VCL only), or TTIcon. Canvas objects can be used outside the *Synchronize* method by locking them.
List	While list objects are not thread-safe, you can use a thread-safe version, TThreadList, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

Using Thread-local Variables

The thread function and any of the routines it calls have their own local variables, just like any other Dephi language routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section (Delphi) or adding the __thread modifier (C++). For example,

```
[Delphi]
threadvar
x : integer;
```

```
[C++]
int __thread x;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

In C++, the following declarations require runtime initialization and are therefore illegal:

```
[C++]
int f( );
int __thread x = f( );   // illegal
```

Instantiation of a class with a user-defined constructor or destructor requires runtime initialization and is therefore illegal:

```
[C++]
class X {
    X( );
    ~X( );
};
X __thread myclass;            // illegal
```

Checking for Termination by Other Threads

Your thread object begins running when the *Execute* method is called (see Executing thread objects) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the Terminated property. When another thread tries to terminate your thread, it calls the Terminate method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your Execute method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
[Delphi]
procedure TMyThread.Execute;
begin
while not Terminated do
```

```
PerformSomeTask;
end;
```

```
[C++]
void __fastcall TMyThread::Execute()
{
while (!Terminated)
PerformSomeTask();
}
```

Handling Exceptions in the Thread Function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a **try...except** block to the implementation of the *Execute* method:

```
[Delphi]
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
  PerformSomeTask;
  except
    { do something with exceptions }
  end;
end;
```

```
[C++]
void __fastcall TMyThread::Execute()
{
  try
  {
    while (!Terminated)
      PerformSomeTask();
  }
  catch (...)
  {
    // do something with exceptions
  }
}
```

Writing Clean-up Code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an OnTerminate event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the Execute method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main VCL thread values).
- You can safely access any objects from the *OnTerminate* event handler without worrying about clashing with other threads.

Coordinating Threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Whether using thread objects or generating threads using *BeginThread*, the following topics describe techniques for coordinating threads:

- Avoiding Simultaneous Access
- Waiting for Other Threads
- Using the Main VCL Thread

When global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads.

Avoiding Simultaneous Access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

Three techniques prevent other threads from accessing the same memory as your thread:

- Locking Objects
- Using Critical Sections
- Using a Multi-read Exclusive-write Synchronizer

Locking Objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a Lock method that prevents other threads from accessing the canvas until the Unlock method is called.

VCL applications also include a thread-safe list object, TThreadList. Calling LockList returns the list object while also blocking other execution threads from using the list until the UnlockList method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

Using Critical Sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of TCriticalSection.

TCriticalSection has two methods, Acquire(which blocks other threads from executing the section) and Release (which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

Warning: Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY*, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
[Delphi]
LockXY.Acquire; { lock out other threads }
try
Y := sin(X);
finally
LockXY.Release;
end;
```

```
[C++]
pLockXY->Acquire(); // lock out other threads
try
{
Y = sin(X);
}
__finally
{
pLockXY->Release();
}
```

Using the Multi-read Exclusive-write Synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using TMultiReadExclusiveWriteSynchronizer. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the BeginRead method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the EndRead method. Any thread that writes to the protected memory must call BeginWrite first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the EndWrite method, so that threads waiting to read the memory can begin.

Warning: Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

Other Techniques for Sharing Memory

When using VCL objects, use the main thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by VCL objects in other threads. See Using the Main VCL Thread for more information on the main thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See Using Thread-local Variables for more information about thread-local variables.

Waiting for Other Threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either

- Wait for another thread to completely finish executing, or
- Wait for a task to be completed.

Waiting for a Thread to Finish Executing

To wait for another thread to finish executing, use the WaitFor method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own Execute method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```
[Delphi]
if ListFillingThread.WaitFor then
begin
with ThreadList1.LockList do
begin
for I := 0 to Count - 1 do
ProcessItem(Items[I]);
end;
ThreadList1.UnlockList;
end;
```

```
[C++]
if (pListFillingThread->WaitFor())
{
  TList *pList = ThreadList1->LockList();
for (int i = 0; i < pList->Count; i++)
ProcessItem(pList->Items[i]);
ThreadList1->UnlockList();
}
```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the ReturnValue property.

ReturnValue is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

Waiting for a Task to Be Completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (TEvent) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. SetEvent turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the ResetEvent method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the WaitFor method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the OnTerminate event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```
[Delphi]
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
...
CounterGuard.Acquire; { obtain a lock on the counter }
Dec(Counter);   { decrement the global counter variable }
if Counter = 0 then
Event1.SetEvent; { signal if this is the last thread }
CounterGuard.Release; { release the lock on the counter }
...
end;
```

```
[C++]
void __fastcall TDataModule::TaskThreadTerminate(TObject *Sender)
{
...
CounterGuard->Acquire(); // lock the counter
if (--Counter == 0)    // decrement the global counter
Event1->SetEvent(); // signal if this is the last thread
CounterGuard->Release(); // release the lock on the counter
}
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the WaitFor method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from the following table:

WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the time-out period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
[Delphi]
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
raise Exception;
{ now continue with the main thread. All task threads have finished }
```

```
[C++]
Event1->ResetEvent(); // clear the event before launching the threads
for (int i = 0; i < Counter; i++)
new TaskThread(false); // create and launch task threads
if (Event1->WaitFor(20000) != wrSignaled)
throw Exception;
// now continue with the main thread, all task threads have finished
```

Note: If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

Executing Thread Objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the Resume method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False*. For example, the following line creates a thread and starts its execution:

```
[Delphi]
SecondThread := TMyThread.Create(false); {create and run the thread }
```

```
[C++]
TMyThread *SecondThread = new TMyThread(false); // create and run the thread
```

Warning: Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

The following topics discuss how to use the threads in your application:

- Overriding the Default Priority.
- Starting and Stopping Threads

Overriding the Default Priority

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in Initializing the thread. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
[Delphi]
SecondThread :=  TMyThread.Create(True); { create but don't run }
SecondThread.Priority := tpLower; { set the priority lower than normal }
SecondThread.Resume; { now run the thread }
```

```
[C++]
TMyThread *SecondThread = new TMyThread(true); // create but don't run
SecondThread->Priority = tpLower; // set the priority lower than normal
SecondThread->Resume(); // now run the thread
```

Starting and Stopping Threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its Suspend method. When it is safe for the thread to resume, call its Resume method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the Terminate method. *Terminate* sets the thread's Terminated property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.

Debugging Multi-threaded Applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose **View** ▸ **Debug Windows** ▸ **Threads** from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

Naming a Thread

Because it is difficult to tell which thread ID refers to which thread in the Thread Status box, you can name your thread classes. When you are creating a thread class in the Thread Object dialog box, besides entering a class name, also check the Named Thread check box, enter a thread name, and click OK.

Naming the thread class adds a method to your thread class called *SetName*. When the thread starts running, it calls the *SetName* method first.

Note: You can name threads in VCL applications only.

You can also:

- Convert an unnamed thread to a named thread.
- Assign separate names to similar threads.

Converting an Unnamed Thread to a Named Thread

You can convert an unnamed thread to a named thread. For example, if you have a thread class that was created using Delphi 6 or earlier, convert it into a named thread.

To convert an unnamed thread to a named thread

1 Add the Windows unit to the **uses** clause of the unit your thread is declared in:

```
[Delphi]
//---------------------------------------------------------------------------
uses
Classes {$IFDEF MSWINDOWS} , Windows {$ENDIF};
//---------------------------------------------------------------------------
```

2 Add the *SetName* method to your thread class in the **interface** section:

```
[Delphi]
//---------------------------------------------------------------------------
type
TMyThread = class(TThread)
private
procedure SetName;
protected
procedure Execute; override;
end;
//---------------------------------------------------------------------------
```

```
[C++]
//---------------------------------------------------------------------------
void TMyThread::SetName()
{
THREADNAME_INFO info;
info.dwType = 0x1000;
info.szName = "MyThreadName";
info.dwThreadID = -1;
info.dwFlags = 0;
__try
{
RaiseException( 0x406D1388, 0, sizeof(info)/sizeof(DWORD),(DWORD*)&info );
}
__except (EXCEPTION_CONTINUE_EXECUTION)
{
}
}
//---------------------------------------------------------------------------
```

3 Add the *TThreadNameInfo* record and *SetName* method in the **implementation** section:

```
[Delphi]
//---------------------------------------------------------------------------
{$IFDEF MSWINDOWS}
type
TThreadNameInfo = record
FType: LongWord;     // must be 0x1000
FName: PChar;        // pointer to name (in user address space)
FThreadID: LongWord; // thread ID (-1 indicates caller thread)
FFlags: LongWord;    // reserved for future use, must be zero
  end;
{$ENDIF}
{ TMyThread }
procedure TMyThread.SetName;
{$IFDEF MSWINDOWS}
var
ThreadNameInfo: TThreadNameInfo;
{$ENDIF}
begin
{$IFDEF MSWINDOWS}
ThreadNameInfo.FType := $1000;
ThreadNameInfo.FName := 'MyThreadName';
ThreadNameInfo.FThreadID := $FFFFFFFF;
ThreadNameInfo.FFlags := 0;
try
RaiseException( $406D1388, 0, sizeof(ThreadNameInfo) div sizeof(LongWord),
@ThreadNameInfo );
 except
  end;
{$ENDIF}
end;
//---------------------------------------------------------------------------
```

Note: Set *TThreadNameInfo* to the name of your thread class.

The debugger sees the exception and looks up the thread name in the structure you pass in. When debugging, the debugger displays the name of the thread in the Thread Status box's thread ID field.

4 Add a call to the new *SetName* method at the beginning of your thread's *Execute* method:

```
[Delphi]
//---------------------------------------------------------------------------
procedure TMyThread.Execute;
begin
SetName;
{ Place thread code here }
end;
//---------------------------------------------------------------------------
```

```
[C++]
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
SetName();
//---- Place existing Execute method code here ----
}
//---------------------------------------------------------------------------
```

Assigning Separate Names to Similar Threads

All thread instances from the same thread class have the same name. However, you can assign a different name for each thread instance at runtime using the following steps.

To assign separate names to similar threads

1 Add a *ThreadName* property to the thread class by adding the following in the class definition:

```
[Delphi]
property ThreadName: string read FName write FName;
```

```
[C++]
__property AnsiString ThreadName = {read=FName, write=FName};
```

2 In the *SetName* method, change where it says:

```
[Delphi]
ThreadNameInfo.FName := 'MyThreadName';
```

```
[C++]
info.szName = "MyThreadName";
```

to:

```
[Delphi]
ThreadNameInfo.FName := ThreadName;
```

```
[C++]
info.szName = ThreadName;
```

To create the thread object

1 Create it suspended. See Executing Thread Objects.
2 Assign a name, such as `MyThread.ThreadName := 'SearchForFiles';`
3 Resume the thread. See Starting and Stopping Threads.

Exception handling

Exception Handling

Exceptions are exceptional conditions that require special handling. They include errors that occur at runtime, such as divide by zero, and the exhaustion of free store. Exception handling provides a standard way of dealing with errors, discovering both anticipated and unanticipated problems, and enables developers to recognize, track down, and fix bugs.

When an error occurs, the program raises an exception, meaning it creates an exception object and rolls back the stack to the first point it finds where you have code to handle the exception. The exception object usually contains information about what happened. This allows another part of the program to diagnose the cause of the exception.

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application presents a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

See Defining Protected Blocks for details on how to create and handle exceptions.

For information on using exceptions with the routines and classes in VCL, see Handling Exceptions in VCL Applications.

Defining Protected Blocks

To prepare for exceptions, you place statements that might raise them in a try block. If one of these statements does raise an exception, control is transferred to an exception handler that handles that type of exception, then leaves the block. The exception handler is said to *catch* the exception and specifies the actions to take. By using try blocks and exception handlers, you can move error checking and error handling out of the main flow of your algorithms, resulting in simpler, more readable code.

You start a protected block using the keyword **try**. The exception handler must immediately follow the try block. It is introduced by the keyword **except**, and signals the end of the try block. This syntax is illustrated in the following code. If the *SetFieldValue* method fails and raises an *EIntegerRange* exception, execution jumps to the exception-handling part, which displays an error message. Execution resumes outside the block.

```
[Delphi]
try
  SetFieldValue(dataField, userValue);
except
  on E: EIntegerRange do
    ShowMessage(Format('Expected value between %d and %d, but got %d',
                       E.Min, E.Max, E.Value));
end;
  .  { execution resumes here, outside the protected block }
  .
  .
```

```
[C++]
try
{
SetFieldValue(dataField, userValue);
}
catch (EIntegerRange &E)
{
  ShowMessage(Format("Expected value between %d and %d, but got %d\n",
              E.Min, E.Max, E.Value));
}
// execution resumes here, outside the protected block
```

You must have an *exception handling block* (described in Writing Exception Handlers) or a *finally block* (described in Writing Finally Blocks) immediately after the try block. An exception handling block should include a handler for each exception that the statements in the try block can generate.

Writing the Try Block

The first part of a protected block is the try block. The try block contains code that can potentially raise an exception. The exception can be raised either directly in the try block, or by code that is called by statements in the try block. That is, if code in a try block calls a routine that doesn't define its own exception handler, then any exceptions raised inside that routine cause execution to pass to the exception-handler associated with the try block. Keep in mind that exceptions don't come just from your code. A call to an RTL routine or another component in your application can also raise an exception.

The following example demonstrates catching an exception thrown from a *TFileStream* object.

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
  fileStream: TFileStream;
begin
     try
        (* Attempt to open a non-existant file *)
        fileStream := TFileStream.Create('NOT_THERE.FILE', fmOpenRead);
                              (* Process the file contents... *)
        fileStream.Free;
     except
        on EFOpenError do ShowMessage('EFOpenError Raised');
     else
        ShowMessage('Exception Raised');
     end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   TFileStream *fileStream;
      try {
// Attempt to open a non-existent file
         fileStream = new TFileStream("NOT_THERE.FILE", fmOpenRead);
// Process the file contents...
         delete fileStream;
      }
      catch(EFOpenError &e) {
         ShowMessage("EFOpenError Raised");
      }
      catch(...) {
         ShowMessage("Exception Raised");
      }
}
```

Using a try block makes your code easier to read. Instead of sprinkling error-handling code throughout your program, you isolate it in exception handlers so that the flow of your algorithms is more obvious.

This is especially true when performing complex calculations involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid. By using exceptions, you can spell out the normal expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every time to make sure you can proceed with each step in the calculation.

For details on raising exceptions from the code in your try block, see Raising an Exception.

Raising an Exception

To indicate a disruptive error condition, you can raise an exception by constructing an instance of an exception object that describes the error condition and calling the reserved word **raise**.

To raise an exception, call the reserved word **raise**, followed by an instance of an exception object. This establishes the exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling raise with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

Raising an exception sets the *ErrorAddr* variable in the System unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the raise clause that appears in *ErrorAddr* when an exception occurs.

Warning: Do not assign a value to *ErrorAddr* yourself. It is intended as read-only.

To specify an error address for an exception, add the reserved word *at* after the exception instance, followed by an address expression such as an identifier.

Writing Exception Handlers

The exception handling block appears immediately after the try block. This block incudes one or more exception handlers. An exception handler provides a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, can be more difficult for the application or the user to correct.

The application executes the statements in and exception handler only if an exception occurs during execution of the statements in the preceding try block. When a statement in the try block raises an exception, execution immediately jumps to the exception handler, where it steps through the specified exception-handling statements, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

The following topics provide details on writing exception handlers:

- Exception-handling Statements
- Handling Classes of Exceptions
- Scope of Exception Handlers
- Reraising Exceptions

Exception-handling Statements

The exception handling block starts with the **except** keyword and ends with the keyword **end**. These two keywords are actually part of the same statement as the try block. That is, both the try block and the exception handling block are considered part of a single **try...except** statement.

Inside the exception handling block, you include one or more exception handlers. An exception handler is a statement of the form

```
on <type of exception> do <statement>;
```

For example, the following exception handling block includes multiple exception handlers for different exceptions that can arise from an arithmetic computation:

```
try
{ calculation statements }
except
  on EZeroDivide do Value := MAXINT;
  on EIntOverflow do Value := 0;
  on EIntUnderflow do Value := 0;
end;
```

Much of the time, as in the previous example, the exception handler doesn't need any information about an exception other than its type, so the statements following **on..do** are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of **on.. do** that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance. For example:

```
on E: EIntegerRange do
  ShowMessage(Format('Expected value between %d and %d', E.Min, E.Max));
```

The temporary variable (E in this example) is of the type specified after the colon (*EIntegerRange* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Warning: Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

You can provide a single default exception handler to handle any exceptions for which you haven't provided specific handlers. To do that, add an else part to the exception-handling block:

```
try
{ statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from any containing block.

Warning: It is not advisable to use this all-encompassing default exception handler. The else clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so using a finally block. For details about finally blocks, see Writing Finally Blocks.

Handling Classes of Exceptions

Exceptions are always represented by classes. As such, you usually work with a hierarchy of exception classes. For example, VCL defines the *ERangeError* exception as a descendant of *EIntError*.

When you provide an exception handler for a base exception class, it catches not only direct instances of that class, but instances of any of its descendants as well. For example, the following exception handler handles all integer math exceptions, including *ERangeError*, *EDivByZero*, and *EIntOverflow*:

```
[Delphi]
try
{ statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

```
[C++]
try
{
  // statements that perform integer math operations
}
catch (EIntError &E)
{
```

```
  // special handling for integer math errors
}
```

You can combine error handlers for the base class with specific handlers for more specific (derived) exceptions. You do this by placing the **catch** statements in the order that you want them to be searched when an exception is thrown. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
[Delphi]
try
{ statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

```
[C++]
try
{
  // statements performing integer math
}
catch (const ERangeError &rangeErr)
{
  // out-of-range handling
}
catch (const EIntError &intErr)
{
  // handling for other integer math errors
}
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

Scope of Exception Handlers

You do not need to provide handlers for every kind of exception in every block. You only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains it (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

Thus, you can nest your exception handling code. That is, you can use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. For example:

```
[Delphi]
try
{ statements }
  try
{ special statements }
  except
    on ESomething do
    begin
{ handling for only the special statements }
    end;
  end;
  { more statements }
```

```
except
  on ESomething do
  begin
    {handling for statements and more statements, but not special statements}
  end;
end;
```

```
[C++]
try
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
  // more statements
}
catch (const ESomething &E)
{
  // handling for statements and more statements, but not special statements
}
```

Note: This type of nesting is not limited to exception-handling blocks. You can also use it with finally blocks (described in Writing Finally Blocks) or a mix of exception-handling and finally blocks.

Reraising Exceptions

Sometimes when you handle an exception locally, you want to augment the handling in the enclosing block, rather than replace it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond. You do this by using the raise command with no arguments. This is called reraising or rethrowing the exception. The following example illustrates this technique:

```
[Delphi]
try
{ statements }
  try
{ special statements }
  except
    on ESomething do
    begin
{ handling for only the special statements }
      raise;{ reraise the exception }
    end;
  end;
except
  on ESomething do ...;{ handling you want in all cases }
end;
```

```
[C++]
try
```

```
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
}
catch (const ESomething &E)
{
  // handling you want in all cases
}
```

If code in the *statements* part raises an *ESomething* exception, only the handler in the outer exception-handling block executes. However, if code in the *special statements* part raises *ESomething*, the handling in the inner exception-handling block executes, followed by the more general handling in the outer exception-handling block. By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

If the handler wants to throw a different exception, it can use the raise or throw statement in the normal way, as described in Raising an Exception.

Writing finally Blocks

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. However, there are times when you do not need to handle the exception, but you do have code that you want to execute after the protected block, even if an exception occurs. Typically, such code handles cleanup issues, such as freeing resources that were allocated before the protected block.

By using finally blocks, you can ensure that if your application allocates resources, it also releases them, even if an exception occurs. Thus, if your application allocates memory, you can make sure it eventually releases the memory, too. If it opens a file, you can make sure it closes the file later. Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:

- Files
- Memory
- Windows resources or widget library resources (Qt objects)
- Objects (instances of classes in your application)

The following event handler illustrates how an exception can prevent an application from freeing memory that it allocates:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
APointer: Pointer;
AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an exception }
```

```
  FreeMem(APointer, 1024);{ this never gets called because of the exception}
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
  int AnInteger = 10/ADividend; // this generates an exception
  free(ptr); // this never gets called because of the exception
}
```

Although most errors are not that obvious, the example illustrates an important point: When an exception occurs, execution jumps out of the block, so the statement that frees the memory never gets called.

To ensure that the memory is freed, you can use a try block with a finally block.

For details on writing finally blocks, see Writing a Finally Block.

Writing a Finally Block

Finally blocks are introduced by the keyword **finally**. They are part of a **try..finally** statement, which has the following form:

```
try
{ statements that may raise an exception}
finally
{ statements that are called even if there is an exception in the try block}
end;
```

In a **try..finally** statement, the application always executes any statements in the finally part, even if an exception occurs in the try block. When any code in the try block (or any routine called by code in the try block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part, which is called the cleanup code. After the finally part executes, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the try block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TObject);
var
APointer: Pointer;
AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an exception }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the exception }
  end;
end;
```

The statements in the finally block do not depend on an exception occurring. If no statement in the try part raises an exception, execution continues through the finally block.

Handling Exceptions in VCL Applications

If you use VCL components or the VCL runtime library in your applications, you need to understand the VCL exception handling mechanism. Exceptions are built into many VCL classes and routines and they are thrown automatically when something unexpected occurs. Typically, these exceptions indicate programming errors that would otherwise generate a runtime error. A limited number of these classes is described in VCL Exception Classes.

The mechanics of handling component exceptions are no different than handling any other type of exception.

If you do not handle the exception, VCL handles it in a default manner. Typically, a message displays describing the type of error that occurred. While debugging your application, you can look up the exception class in online Help. The information provided will often help you to determine where the error occurred and its cause.

Certain classes of exceptions do not display an error message when caught by the default handlers. These are described in Silent Exceptions.

A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a "List index out of bounds" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption :=

ListBox1.Items[3]

;{ set form caption to fourth string }
  except
    on EStringListError do
      ShowMessage('List box contains fewer than four strings');
  end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  ListBox1->Items->Add("a string"); // add a string to list box
  ListBox1->Items->Add("another string"); // add another string ...
  ListBox1->Items->Add("still another string"); // ... and a third string
  try
    Caption =
ListBox1->Items->Strings[3]
; // set form caption to 4th string
  catch (EStringListError &E)
    ShowMessage("List box contains fewer than four strings");
}
```

If you click the button once, the list box has only three strings, so accessing the fourth string raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

In addition to handling the exceptions that VCL raises, you can define and raise your own VCL-based exception classes. This is discussed in Defining Your Own VCL Exceptions.

VCL Exception Classes

VCL includes a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions. All VCL exception classes descend from one root object called Exception. *Exception* provides a consistent interface for applications to handle exceptions. It provides the string for the message that VCL exceptions display by default.

The following table lists a selection of the exception classes defined in VCL:

Selected exception classes

Exception class	Description
EAbort	Stops a sequence of events without displaying an error message dialog box.
EAccessViolation	Checks for invalid memory access errors.
EBitsError	Prevents invalid attempts to access a Boolean array.
EComponentError	Signals an invalid attempt to register or rename a component.
EConvertError	Indicates string or object conversion errors.
EDatabaseError	Specifies a database access error.
EDBEditError	Catches data incompatible with a specified mask.
EDivByZero	Catches integer divide-by-zero errors.
EExternalException	Signifies an unrecognized exception code.
EInOutError	Represents a file I/O error.
EIntOverflow	Specifies integer calculations whose results are too large for the allocated register.
EInvalidCast	Checks for illegal typecasting.
EInvalidGraphic	Indicates an attempt to work with an unrecognized graphic file format.
EInvalidOperation	Occurs when invalid operations are attempted on a component.
EInvalidPointer	Results from invalid pointer operations.
EMenuError	Involves a problem with menu item.
EOleCtrlError	Detects problems with linking to ActiveX controls.
EOleError	Specifies OLE automation errors.
EPrinterError	Signals a printing error.
EPropertyError	Occurs on unsuccessful attempts to set the value of a property.
ERangeError	Indicates an integer value that is too large for the declared type to which it is assigned.
ERegistryException	Specifies registry errors.
EZeroDivide	Catches floating-point divide-by-zero errors.

There are other times when you will need to create your own exception classes to handle unique situations. You can declare a new exception class by making it a descendant of type *Exception* and creating as many constructors as you need (or copy the constructors from an existing class in the SysUtils unit).

Default Exception Handling in VCL

If your application code does not catch and handle the exceptions that are raised, the exceptions are ultimately caught and handled by the *HandleException* method of the global *Application* object. For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. If there is no *OnException* event handler (and

the exception is not *EAbort*), *HandleException* displays a message box with the error message associated with the exception.

There are certain circumstances where *HandleException* does not get called. Exceptions that occur before or after the execution of the application's Run method are not caught and handled by *HandleException*. When you write a callback function or a library (.dll or shared object) with functions that can be called by an external application, exceptions can escape the *Application* object. To prevent exceptions from escaping in this manner, you can insert your own call to the *HandleException* method:

```
[Delphi]
try
{ special statements }
except
  on Exception do
  begin
    Application.HandleException(Self);{ call HandleException }
  end;
end;
```

```
[C++]
try
{
  // special statements
}
catch (Exception &E)
{
  Application->HandleException(this);
}
```

Warning: Do not call *HandleException* from a thread's exception handling code.

Silent Exceptions

VCL applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for VCL applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

Note: For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which breaks out of the current operation without displaying an error message.

Note: There is a distinction between *Abort* and *abort*. *abort* kills the application.

The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
I, J: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
    for J := 1 to 10 do {loop ten times }
    begin
      ListBox1.Items.Add(IntToStr(I) + IntToStr(J));
      if I = 7 then Abort;{ abort after the 7th iteration of outer loop}
    end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  for (int i = 1; i <= 10; i++) // loop ten times
    for (int j = 1; j <= 10; j++) // loop ten times
    {
      ListBox1->Items->Add(IntToStr(i) + IntToStr(j));
      if (i == 7)
        Abort(); // abort after 7th iteration of outer loop
    }
}
```

Note that in this example, *Abort* causes the flow of execution to break out of both the inner and outer loops, not just the inner loop.

Defining Your Own VCL Exceptions

Because VCL exceptions are classes, defining a new kind of exception is as simple as declaring a new class type. Although you can raise any object instance as an exception, the standard VCL exception handlers handle only exceptions that descend from *Exception*.

New exception classes should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by an exception handler specific to that exception, one of the standard handlers will handle it instead.

For example, consider the following declaration:

```
[Delphi]
type
  EMyException = class(Exception);
```

```
[C++]
class EMyException : public Exception
{
};
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

Throwing An Exception (C++)

To raise an exception in C++, use the throw keyword. Objects in C++ can be thrown by value, or pointer:

```
// throw an object, to be caught by value or reference
throw EIntegerRange(0, 10, userValue);
// throw an object to be caught by pointer
throw new EIntegerRange(0, 10, userValue);
```

Tip: Throw exceptions by value and catch exceptions by reference to prevent memory leaks. If you catch an exception by pointer, you may not be able to delete the exception object.

Note: To throw an exception by value, it must have a public copy constructor and public destructor.

In addition, the throw statement can throw other types as well. Although it is not recommended, C++ lets you throw primitive types, such as integers or pointers:

```
throw 1;        // throw an int
throw "catastrophic error";            // throw a char *
```

In most cases, you want to throw exception objects because they can provide a more complete description of an error.

Constructors In Exception Handling (C++)

Class constructors can throw exceptions if they cannot successfully construct an object. If a constructor throws an exception, that object's destructor is not necessarily called. Destructors are called only for the base classes and for those objects that were fully constructed inside the classes since entering the try block.

Note: This does not apply to VCL base classes.

Writing A finally Block

bcc32 includes extensions to the C++ language that let it use finally blocks as well. Like exception handlers, a finally block must appear directly after the try block, but it is introduced by the__finally keyword instead of **catch**.

```
try
{
  // statements that may raise an exception
}
__finally
{
    // statements that are called even if there is an exception in the try block
}
```

The application always executes any statements in the finally part, even if an exception occurs in the try block. When any code in the try block (or any routine called by code in the try block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part. After the finally part executes, the exception handler is called. If no exception occurs, the code in the finally block executes in the normal order, after all the statements in the try block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
  try
    int AnInteger = 10/ADividend; // this generates an exception
  __finally
    free(ptr); // this gets called anyway, despite the exception
}
```

The statements in the finally block do not depend on an exception occurring. If no statement in the try part raises an exception, execution continues through the finally block.

Note: Traditional C++ code does not include support for finally block. Instead, it tends to use destructors to handle the freeing of resources. However, when working with VCL, which is written in Delphi, finally blocks are an important tool because of the way VCL objects must be allocated on the heap.

Tip: You can also use `std::auto_ptr` to ensure that allocated objects are deleted.

Unwinding Exceptions (C++)

When an exception is thrown, the runtime library takes the thrown object, gets the type of the object, and looks upward in the call stack for a handler whose type matches the type of the thrown object. Once a handler is found, the RTL unwinds the stack to the point of the handler, and executes the handler.

In the unwind process, the RTL calls destructors for all local objects in the stack frames between where the exception was thrown and where it is caught. If a destructor causes an exception to be raised during stack unwinding and does not handle it, terminate is called. Destructors are called by default, but you can switch off the default by using the -xd compiler option.

Note: During the unwind process, the RTL does not call destructors for objects that are allocated on the heap rather than the stack. This is why, for example, VCL applications use finally blocks to ensure that VCL objects, which are always allocated on the heap, are properly freed. There is one exception to this rule, which is the use of safe pointers.

Smart Pointers (C++)

If you have local variables that are pointers to objects and an exception is thrown, these pointers are not automatically deleted. This is because there is no good way for the compiler to distinguish between a pointer to data that was allocated for this function only and any other pointer. The class that you can use to ensure that objects allocated for local use are destroyed in the even of an exception is `auto_ptr`. There is a special case in which memory is freed for a pointer allocated in a function:

```
auto_ptr< TMyObject > pMyObject( new TMyObject );
```

In this example, if the constructor for `TMyObject` throws an exception, then the pointer to the object allocated for TMYObject will be deleted by the RTL when it unwinds the exception. This is the only time that the compiler automatically deletes a pointer value for you.

Exception Handling Options (C++)

The following exception handling options are available for bcc32:

-x	Enables C++ exception handling. Enabled by default.
-xd	Enables destructor cleanup. Calls destructors for all automatically declared objects between the scope of the catch and throw statements when an exception is thrown. Enabled by default.
-xp	Enables the program to use the `__ThrowFileName` global to obtain the file where the exception occurred and the `__ThrowLineNumber` global to access the line number from where the C++ exception was thrown. Disabled by default.

Working with packages and components

Working with Packages and Components: Overview

A *package* is a special dynamic-link library used by applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time* packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used VCL components reside in a package called vcl . Each time you create a new default VCL application, it automatically uses vcl. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in the runtime package called vcl90.bpl. A computer with several package-enabled applications installed on it needs only a single copy of vcl90.bpl, which is shared by all the applications and the IDE itself.

Several runtime packages encapsulate VCL components while several design-time packages manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Delphi components, you can compile your components into design-time packages before installing them.

Why Use Packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller, saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

Packages and Standard DLLs

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages:

Package files

File extension	Contents
.bpf	A source file required for a package.
bpi	A Borland package import library. A .bpi is created for each package. The bpis for bpls is analogous to import libraries for dlls. This file is passed to the linker by applications using the package to resolve references to functions in the package. The base name for the bpi is the base name for the package source file.
bpk and bpkw	The project options source file. This file is the XML portion of the package project. The ProjectName.bpk and ProjectName.cpp combined are used to manage settings, options, and files used by the package project. .bpk and .bpkw packages are identical, but use the .bpkw extension for packages that you want to use in cross-platform applications.
bpl	The runtime package. This file is a Windows .dll with special -specific features. The base name for the .bpl is the base name of the .bpk or .bpkw source file.
cpp	ProjectName.cpp contains the entry point for the package. Additionally, each component contained within the package generally resides within a .cpp file.
h	The header file or interface for the component. C omponentN ame.h is the companion to C omponentN ame. cpp.
lib	A static library, or collection of .objs , used in place of a .bpi when the application does not use runtime packages. Generated only if the **-Gl** (Generate .lib file) linker option is selected.
obj	A binary image for a unit file contained in a package. One .obj is created, when necessary, for each .cpp file.

Note: Packages share their global data with other modules in an application.

Runtime Packages

Runtime packages are deployed with your applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages (.bpl files) that the application uses. The .bpl files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .bpls.

- Loading packages in an application
- Deciding which runtime packages to use
- Custom packages

Loading Packages in an Application

You can dynamically load packages by either:

- Choosing Project Options dialog box in the IDE; or
- Using the LoadPackage function.

To load packages using the >Project>Options dialog box

1. Load or create a project in the IDE.
2. Choose **Project** ▸ **Options**.
3. Choose the Packages tab.

4 Select the Build with Runtime Packages check box, and enter one or more package names in the edit box underneath. Each package is loaded implicitly only when it is needed (that is, when you refer to an object defined in one of the units in that package). (Runtime packages associated with installed design-time packages are already listed in the edit box.)

5 To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

   If you edit the Search Path edit box in the Add Runtime Package dialog, you can change the global Library Path.

   You do not need to include file extensions with package names (or the version number representing the Delphi release); that is, vcl90.bpl in a VCL application is written as vcl. If you type directly into the Runtime Package edit box, be sure to separate multiple names with semicolons. For example:

```
rtl;vcl;vcldb;vclado;

vclbde;
```

   Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the Build with runtime packages check box is unchecked, the application is compiled without packages.

   Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the Defaults check box at the bottom of the dialog.

Note: When you create an application with packages, you must include the names of the original Delphi units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
[Delphi]
unit MainForm;
interface
uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs;
```

```
[C++]
#include "vcldb.h"
```

The units referenced in this VCL example are contained in the vcl and rtl packages. Nonetheless, you must keep these references in the **uses** clause, even if you use vcl and rtl in your application, or you will get compiler errors. In generated source files, the Form Designer adds these units to the uses clause automatically.

Loading Packages with the LoadPackage Function

You can also load a package at runtime by calling the *LoadPackage* function. *LoadPackage* loads the package specified by its name parameter, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
[Delphi]
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

```
[C++]
if (OpenDialog1->Execute())
  PackageList->Items->AddObject(OpenDialog1->FileName, (TObject *)LoadPackage(OpenDialog1->FileName));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

Deciding Which Runtime Packages to Use

Several runtime packages, including rtl and vcl (VCL application) , supply basic language and component support. The vcl (VCL) package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need several runtime packages, including vcl, vcldb, rtl, and dbrtl (VCL) . If you want to use visual components in your application, you also need vclx (VCL) . To use these packages, choose **Project** ▸ **Options**, select the Packages tab, and make sure the following list is included in the Runtime Packages edit box. You need netclx for Web server applications, as well as baseclx and probably visualclx.

```
vcl;rtl;vcldb;vclx; //For VCL applications
```

Note: In VCL applications, you don't have to manually include vcl and rtl, because they are referenced in the Requires clause of vcldb. Your application compiles just the same whether or not vcl and rtl are included in the Runtime Packages edit box.

Another way you can determine which packages are called by an application is to run it then review the event log (choose **View** ▸ **Debug Windows** ▸ **Event Log**). The event log displays every module that is loaded including all packages. The full package names are listed. So, for example, for vcl90.bpl , you would see a line similar to the following in a VCL application:

```
Module Load: vcl90.bpl Has Debug Info. Base Address $400B0000. Process Project1.exe ($22C)
```

Custom Packages

A custom package is either a .bpl you code and compile yourself or an existing package from a third-party vendor. To use a custom runtime package with an application, choose **Project** ▸ **Options** and add the name of the package to the Runtime Packages edit box on the Packages page.

For example, suppose you have a statistical package called stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
vcl;rtl;vcldb;stats //In VCL applications
```

If you create your own packages, add them to the list as needed.

Design-time Packages

Design-time packages are used to install components on the IDE's **Tool palette** and to create special property editors for custom components. Which ones are installed depends on which edition of Delphi you are using and

whether or not you have customized it. You can view a list of what packages are installed on your system by choosing **Component** ▸ **Installed .NET Components**.

The design-time packages work by calling runtime packages, which they reference in their Requires clause. For example, dclstd references vcl. The dclstd itself contains additional functionality that makes many of the standard components available on the **Tool palette**.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The dclusr design-time package is provided as a default container for new components. See Installing Component Packages

Installing Component Packages

All components are installed in the IDE as packages. If you've written your own components, create and compile a package that contains them. Your component source code must follow the model described in Overview of component creation.

To install or uninstall your own components, or components from a third-party vendor

1. If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with .bpl, .dcp, and .dcu files, be sure to copy all of them.

   The directory where you store the .dcp file—and the .dcu files, if they are included with the distribution—must be in the Delphi Library Path.

   If the package is shipped as a .dpc (package collection) file, only the one file needs to be copied; the .dpc file contains the other files. (For more information about package collection files, see Package collection files.)

2. Choose **Component** ▸ **Install Packages** from the IDE menu, or choose **Project** ▸ **Options** and click the Packages tab. A list of available packages appears in the Design packages list box.

   - To install a package in the IDE, select the check box next to it.
   - To uninstall a package, uncheck its check box.
   - To see a list of components included in an installed package, select the package and click Components.
   - To add a package to the list, click Add and browse in the Add Design Package dialog for the directory where the .bpl file resides (see step 1). Select the .bpl or .dpc file and click Open. If you select a .dpc file, a new dialog box appears to handle the extraction of the .bpl and other files from the package collection.
   - To remove a package from the list, select the package and click Remove.

3. Click OK.

The components in the package are installed on the **Tool palette** pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the ddefault settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the Packages tab of the Project Options dialog box.

To remove components from the **Tool palette** without uninstalling a package, right-click the component to invoke the context menu and choose Hide "<ComponentName>" Button.

Creating and Editing Packages

Creating a package involves specifying:

- A *name* for the package.

- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units . The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Developer Studio 2006 generates a package source file (.dpk).

- Creating a Package
- Editing an Existing Package
- Editing Package Source Files Manually
- Understanding the Structure of a Package
- Compiling packages

Creating a Package

Refer to Understanding the structure of a package for more information about the steps outlined here.

To create a package

1. Choose **File** ▸ **New** ▸ **Other**, select the Package icon under Delphi Projects, and click OK. The generated package appears in the **Project Manager**. The **Project Manager** displays a *Requires* node and a *Contains* node for the new package.
2. To add a unit to the **contains** clause, right-click the contains node in the **Project Manager** and select Add. In the Add Unit page, type a .pas file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the **Project Manager**. You can add additional units by repeating this step.
3. To add a package to the **requires** clause, right-click the requires node in the **Project Manager** and select Add Reference. In the Requires page, type a .dcp file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the **Project Manager**. You can add additional packages by repeating this step.
4. In the **Project Manager**, right-click your package and select Compile.

Editing an Existing Package

To edit an existing package:

1. Choose **File** ▸ **Open** (or **File** ▸ **Reopen**) and select a dpk file.
2. In the Project Manager, select one of the packages in the Requires node, right-click, and choose Open.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the defproj.dof file.

Understanding the Structure of a Package

Packages include the following parts:

- Package name
- Requires clause
- Contains clause

Naming packages

Package names must be unique within a project. If you name a package Stats, Developer Studio 2006 generates a source file for it called Stats.dpk; the compiler generates an executable and a binary image called Stats.bpl and Stats.dcp, respectively. Use Stats to refer to the package in the **requires** clause of another package, or when using the package in an application.

Requires clause

The **requires** clause specifies other, external packages that are used by the current package. An external package included in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units.'

Note: Most packages that you create require rtl. If using VCL components, you'll also need to include the vcl package.

Avoiding circular package references

Packages cannot contain circular references in their requires clause. This means that:

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Handling duplicate package references

Duplicate references in a package's **requires** clause—or the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

Contains clause

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in pas files and include them in the **contains** clause.

Avoiding redundant source code uses

A package cannot appear in the containsclause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, including the IDE. This means that if you create a package that contains one of the units in vcl (VCL) you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.

Editing Package Source Files Manually

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Del source code.

To open a package source file in the Code editor

1. Open the package in Developer Studio 2006.
2. Right-click the package in the **Project Manager** and choose View Source.

- The **package** heading specifies the name for the package.
- The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a requires clause.
- The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning).

For example, the following VCL code declares the vcldb package (in the source file vcldb90.bpl):

```
package MyPack;
{$R *.res}
    ...{compiler directives omitted}
requires
    rtl,
    vcl;
contains
    Db,
    NewComponent1 in 'NewComponent1.pas';
end.
```

Compiling Packages

You can compile a package from the IDE or from the command line.

To recompile a package by itself from the IDE

1. Choose **File** ▸ **Open** and select a package (.dpk or .dpkw).
2. Click Open.
3. When the package opens:

- In the **Project Manager**, right-click the package and choose Compile.

- In the IDE, choose **Project** ▸ **Build**.

Note: Right-click the package project nodes for options to compile or build.

You can insert compiler directives into your package source code.

If you compile from the command line, you can use several package-specific switches.

- Package-specific Compiler Directives
- Weak Packaging
- Using the Command-line Compiler and Linker
- Package Files Created by a Successful Compilation

Package-specific Compiler Directives

The following table lists package-specific compiler directives that you can insert into your source code.

Package-specific compiler directives

Directive	Purpose
#pragma package(smart_init)	Assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.)
#pragma package(smart_init, weak)	Packages unit "weakly." See (Put directive in unit source file.)

Note: Including **{$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{$G-}** or **{$IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler Directives for information on compiler directives not discussed here.

See Package-specific Compiler Directives.

Refer to Creating Packages and DLLs for additional directives that can be used in all libraries.

Weak Packaging

The **$WEAKPACKAGEUNIT** directive affects the way a .dcu file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see Package files created when compiling.) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be *weakly packaged*.

For example, suppose you've created a package called pack1 that contains only one unit, unit1. Suppose unit1 does not use any additional units, but it makes calls to rare.dll. If you put the **{$WEAKPACKAGEUNIT ON}** directive in unit1.pas (Delphi) or unit1.cpp (C++) when you compile your package, unit1 will not be included in pack1.bpl; you will not have to distribute copies of rare.dll with pack1. However, unit1 will still be included in pack1.dcp. If unit1 is referenced by another package or application that uses pack1, it will be copied from pack1.dcp and compiled directly into the project.

Now suppose you add a second unit, unit2, to pack1. Suppose that unit2 uses unit1. This time, even if you compile pack1 with **{$WEAKPACKAGEUNIT ON}** in unit1.pas, the compiler will include unit1 in pack1.bpl. But other packages or applications that reference unit1 will use the (non-packaged) copy taken from pack1.dcp.

Note: Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **{$WEAKPACKAGEUNIT ON}** directive is an advanced feature intended for developers who distribute their packages to other programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, the PenWin unit references PenWin.dll. Most projects don't use PenWin, and most computers don't have PenWin.dll installed on them. For this reason, the PenWin unit is weakly packaged in vcl. When you compile a project that uses PenWin and the vcl package, PenWin is copied from vcl70.dcp and bound directly into your project; the resulting executable is statically linked to PenWin.dll.

If PenWin were not weakly packaged, two problems would arise. First, vcl itself would be statically linked to PenWin.dll, and so you could not load it on any computer which didn't have PenWin.dll installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both vcl and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of vcl.

Compiling and Linking from the Command Line

When you compile from the command line, you can use the package-specific switches listed in the following table.

Package-specific command-line compiler switches

Switch	Purpose
-$G-	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE*path*	Specifies the directory where the package file (.bpl) will be placed.
-LN*path*	Specifies the directory where the package file (.dcp) will be placed.
-LUpackage	Use packages.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Note: Using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See The Command-line Compiler for information on command-line options not discussed here.

Package-specific command-line compiler and linker switches

Switch	Purpose
tP	Generates a project as a package (compiler switch).
-D "*description*"	Saves the specified description with the package.
-Gb	Generates the .bpl filename.
-Gi	Saves the generated .bpi file. Included by default in package project files.
-Gpd	Generates a design-time-only package.
-Gpr	Generates a runtime-only package.
-Gl	Generates a .lib file.
-Tpp	Builds the project as a package. Included by default in package project files.

Package Files Created by Compiling

To create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called traypak.bpl.

```
<PROJECT value="Traypak.bpl"/>
```

A successfully compiled package includes .dcp, .dcu and bpl files. For a detailed description of these files, see Packages and standard DLLs.

These files are generated by default in the directories specified in Library page of the **Tools** ▸ **Options** ▸ **Environment Options** ▸ **Delphi Options** ▸ **Library** dialog. You can override the default settings by right-clicking the package in the **Project Manager** and choosing Options to display the Project Options dialog; make any changes on the Directories/Conditionals page.

Deploying Packages

You deploy packages much like you deploy other applications. The files you distribute with a deployed package may vary. The bpl and any packages or dlls required by the bpl must be distributed.

Files deployed with a package

File	Description
ComponentName.h	Allows end users access to the class interfaces.
ComponentName.cpp	Allows end users access to the component source.
.bpi, .obj, and .lib	Allows end users to link applications.

For general deployment information, refer to Deploying applications.

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application's .exe file as well as all the library (.bpl or .dll) files that the application calls. If the library files are in a different directory from the .exe file, they must be accessible through the user's Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallShield Express, your installation script can check the user's system for any packages it requires before blindly reinstalling them.

Distributing packages to other developers

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .dcp and .bpl files. You will probably want to include .dcu files as well.

Package Collection Files

Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create package collection files

1. Choose **Tools** ▸ **Package Collection Editor** to open the Package Collection editor.
2. Either choose **Edit** ▸ **Add Package** or click the Add a package button, then select a bpl in the Select Package dialog and click Open. To add more bpls to the collection, click the Add a package button again. A tree diagram on the left side of the Package editor displays the bpls as you add them. To remove a package, select it and either choose **Edit** ▸ **Remove Package** or click the Remove the selected package button.
3. Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields appear:
   - In the Author/Vendor Name edit box, you can enter optional information about your package collection that appear in the Installation dialog when users install packages.
   - Under Directory list, list the default directories where you want the files in your package collection to be installed. Use the Add, Edit, and Delete buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter Source as a Directory name with C:\MyPackage\Source as the Suggested path. The Installation dialog box will display C:\MyPackage\Source as the suggested path for the directory.
4. In addition to bpls, your package collection can contain .dcp, .dcu, and .pas (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (bpls); the files in a group are installed only when their associated bpl is installed. To place ancillary files in your package collection, select a bpl in the tree diagram and click the Add a file group button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the Package Collection editor.
   - In the Install Directory list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under Directory list in step 3, above.
   - Check the Optional Group check box if you want installation of the files in this group to be optional.
   - Under Include Files, list the files you want to include in this group. Use the Add, Delete, and Auto buttons to edit the list. The Auto button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the Package Collection editor uses the global Library Path to search for these files.
5. You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a bpl in the tree diagram, four new fields appear on the right side of the Package Collection editor:
   - In the Required Executables list box, select the directory where you want the .bpl files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory list in step 3, above.) The Package Collection editor searches for these files using Delphi's global Library Path and lists them under Required Executable Files.
   - In the Required Libraries list box, select the directory where you want the .dcp files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection editor searches for these files using the global Library Path and lists them under Required Library Files.
6. To save your package collection source file, choose **File** ▸ **Save**. Package collection source files should be saved with the .pce extension.
7. To build your package collection, click the Compile button. The Package Collection editor generates a .dpc file with the same name as your source (.pce) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing .pce file, select **File** ▸ **Open** in the Package Collection editor and locate the file you want to work with.

Creating international applications

Creating International Applications: Overview

This topic discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

The following topics are discussed in this section:

- Internationalization and localization
- Internationalizing applications
- Localizing applications

Internationalization and Localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

If your edition includes the Translation Tools, you can use the them to manage localization.

Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports many locales, each of which is described by a language and country pair.

Localization

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified for the tax laws in different countries.

Internationalizing Applications

You need to complete the following steps to create internationalized applications:

- Enable your code to handle strings from international character sets.
- Design your user interface to accommodate the changes that result from localization.
- Isolate all resources that need to be localized.

Enabling Application Code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales. To do this, you must consider the following:

- Character sets
- OEM and ANSI character sets
- Multibyte character sets
- Wide characters
- Locale-specific features

Character Sets

The Western editions (including English, French, and German) of Windows use the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes.

There are generally three types of characters sets:

- Single-byte
- Multibyte
- Wide characters

Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many western operating systems is a single-byte character set.

In a multibyte character set, some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages.

OEM and ANSI Character Sets

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

Multibyte Character Sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the single-

byte *char*. Instead, a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate one-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into two or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters.

Delphi provides you with many of these runtime library functions, as listed in the following table:

Runtime library functions

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

Wide Characters

One approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Unicode characters and strings are also called wide characters and wide character strings. In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words.

The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi supports UCS-2 on both platforms. Because wide characters are two bytes instead of one, the character set can represent many more different characters.

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

The biggest disadvantage of working with wide characters is that Windows supports a few wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require additional code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

Including Bi-directional Functionality in Applications

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

TApplication has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties.

Note: Bi-directional properties are not available for cross-platform applications.

The BiDiMode property controls the reading order for the text, the placement of the vertical scrollbar, and whether the alignment is changed. Controls that have a text property, such as *Name*, display the *BiDiMode* property on the **Object Inspector**.

The *BiDiMode* property is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly*.

bdLeftToRight

bdLeftToRight draws text using left to right reading order. The alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

bdRightToLeft

bdRightToLeft draws text using right to left reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left to right.

bdRightToLeftNoAlign

bdRightToLeftNoAlign draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

bdRightToLeftReadingOnly

bdRightToLeftReadingOnly draws text using right to left reading order, and the alignment and scroll bars are not changed.

ParentBiDiMode Property

ParentBiDiMode is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from Application. If all the

ParentBiDiMode properties are *True*, when you change Application's *BiDiMode* property, all forms and controls in the project are updated with the new setting.

FlipChildren Method

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox.FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the Left property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped the edit control's right side is five pixels from the right edge of the parent control. If the edit control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select **Edit** ▸ **Flip Children** and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

Note: Selecting an edit control and issuing a Flip Children|Selected command does nothing. This is because edit controls are not containers.

Additional Methods

There are several other methods useful for developing applications for bi-directional users.

VCL methods that support BiDi

Method	Description
OkToChangeFieldAlignment	Used with database controls. Checks to see if the alignment of a control can be changed.
DBUseRightToLeftAlignment	A wrapper for database controls for checking alignment.
ChangeBiDiModeAlignment	Changes the alignment parameter passed to it. No check is done for *BiDiMode* setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
IsRightToLeft	Returns *True* if any of the right to left options are selected. If it returns *False* the control is in left to right mode.
UseRightToLeftReading	Returns *True* if the control is using right to left reading.
UseRightToLeftAlignment	Returns *True* if the control is using right to left alignment. It can be overridden for customization.
UseRightToLeftScrollBar	Returns *True* if the control is using a left scroll bar.
DrawTextBiDiModeFlags	Returns the correct draw text flags for the *BiDiMode* of the control.
DrawTextBiDiModeFlagsReadingOnly	Returns the correct draw text flags for the *BiDiMode* of the control, limiting the flag to its reading order.
AddBiDiModeExStyle	Adds the appropriate *ExStyle* flags to the control that is being created.

Locale-specific Features

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

Controls offer support in programming the IME. Most windowed controls that work directly with text input have an ImeName property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an ImeMode property that specifies how the IME should convert keyboard input. ImeName introduces several protected methods that you can use to control the IME from classes you define. In addition, the global Screen variable provides information about the IMEs available on the user's system.

The global *Screen* variable also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

The IME is available in VCL applications only.

Designing the User Interface

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

The following topics are discussed in this section:

- Text
- Graphic images
- Formats and sort order
- Keyboard mappings

Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. The following table provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Estimating string lengths

Length of English String (in characters)	Expected Increase
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Graphic Images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not

appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

Formats and Sort Order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, two-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

Keyboard Mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

Isolating Resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .dfm file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the *Delphi Language Guide*. It is best to include all resource strings in a single, separate unit.

For information on using resource DLLs in your applications see Creating Resource DLLs and Using Resource DLLs.

Creating Resource DLLs

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code to obtain the locale code for the target translation:

```
[Delphi]
unit locales;
interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;
{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  with Languages do
  begin
    for I := 0 to Count - 1 do
    begin
      ListBox1.Items.Add(Name[I]);
    end;
  end;
end;
```

```
[C++]
/* This callback fills a listbox with the strings and their associated languages and
countries*/
BOOL __stdcall EnumLocalesProc(char* lpLocaleString)
{
AnsiString LocaleName, LanguageName, CountryName;
LCID lcid;
lcid = StrToInt("$" + AnsiString(lpLocaleString));
LocaleName = GetLocaleStr(lcid, LOCALE_SABBREVLANGNAME, "");
LanguageName = GetLocaleStr(lcid, LOCALE_SNATIVELANGNAME, "");
CountryName = GetLocaleStr(lcid, LOCALE_SNATIVECTRYNAME, "");
if (lstrlen(LocaleName.c_str()) > 0)
```

```
Form1->ListBox1->Items->Add(LocaleName + ":" + LanguageName + "-" + CountryName);
return TRUE;
}
/* This call causes the callback to execute for every locale */
EnumSystemLocales((LOCALE_ENUMPROC)EnumLocalesProc, LCID_SUPPORTED);
```

Using Resource DLLs

The executable, DLLs, and packages (bpls) that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your executable, DLL, or package files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY_CURRENT_USER\Software \Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading applications:

```
[Delphi]
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
Reg: TRegistry;
begin
Reg := TRegistry.Create;
try
if Reg.OpenKey('Software\Borland\Locales', True) then
Reg.WriteString(LocalOverride, FileName);
finally
Reg.Free;
  end;
end;
```

```
[C++]
void SetLocalOverrides(char* FileName, char* LocaleOverride)
{
HKEY Key;
const char* LocaleOverrideKey = "Software\\Borland\\Locales";
if (RegOpenKeyEx(HKEY_CURRENT_USER, LocaleOverrideKey, 0, KEY_ALL_ACCESS, &Key)
== ERROR_SUCCESS) {
if (lstrlen(LocaleOverride) == 3)
RegSetValueEx(Key, FileName, 0, REG_SZ, (const BYTE*)LocaleOverride, 4);
RegCloseKey(Key);
}
}
```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
[Delphi]
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

```
[C++]
LoadString(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, sizeof(szQuery));
```

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLs.

Dynamic Switching of Resource DLLs

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the ReInit unit in your **uses** statement. (ReInit is located in the Richedit sample in the Demos directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*.

For example, the following code switches the interface language to French:

```
[Delphi]
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

```
[C++]
const  FRENCH = MAKELANGID(SUBLANG_FRENCH, LANG_FRENCH);
if (LoadNewResourceModule(FRENCH))
  ReinitializeForms();
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or re-acquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

Note: Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to the their startup state, apart from differences due to localization.

Localizing Applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Localizing resources

Ideally, your resources have been isolated into a resource DLL that contains form files (.dfm in VCL applications) and a resource file. You can open your forms in the IDE and translate the relevant properties.

Note: In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the **Object Inspector** to display only Localizable properties; to do so, right-click in the **Object Inspector** and use the View menu to filter out unwanted property categories.

You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the **Project Manager**.

Deploying applications

Deploying Applications: Overview

Once your application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe these steps when deploying the following applications:

- Deploying General Applications
- Deploying Database Applications
- Deploying Web Applications
- Programming for Varying Host Environments
- Software License Requirements

Deploying General Applications

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. Nearly all types of applications include the following issues:

- Using installation programs
- Identifying application files
- Helper applications
- DLL locations

Database and Web applications require additional installation steps. For additional information on installing database applications, see Deploying database applications. For more information on installing Web applications, see Deploying Web applications.

Using Installation Programs

Simple applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.

InstallShield Express is a setup toolkit that is bundled with Delphi. InstallShield Express is certified for use with Delphi and the Borland Database Engine. It is based on Windows Installer (MSI) technology.

InstallShield Express is not automatically installed when Delphi is installed, so it must be manually installed if you want to use it to create installation programs. Run the installation program from the Delphi CD to install InstallShield Express. For more information on using InstallShield Express to create installation programs, see the InstallShield Express online help.

Other setup toolkits are available. However, if deploying BDE database applications, you should only use toolkits based on MSI technology and those which are certified to deploy the Borland Database Engine.

Identifying Application Files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files, listed by file name extension
- Package files
- Merge modules
- ActiveX controls

Application Files, Listed by File Name Extension

The following types of files may need to be distributed with an application.

Application files

Type	File Name Extension
Program files	.exe and .dll
Package files	.bpl and .dcp
Help files	.hlp, .cnt, and .toc (if used) or any other Help files your application supports
ActiveX files	.ocx (sometimes supported by a DLL)
Local table files	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

Package Files

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See Merge modules for details.

Borland recommends installing the runtime package files supplied by Borland in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .bpl files need to be distributed.

If you are distributing packages to other developers, supply the .bpl and .dcp files.

Merge Modules

InstallShield Express 3.0 is based on Windows Installer (MSI) technology. With MSI-based setup tools such as InstallShield Express, you can use merge modules for deploying runtime packages. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file.

The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool automatically adds or reports a dependency on one or more other packages. For example, if you add the VCLInternet merge module to an install project, the install tool should also automatically add or report a dependency on the VCLDatabase and StandardVCL modules.

The dependencies for each merge module are listed in the table below. The various install tools may react to these dependencies differently. The InstallShield for Windows Installer automatically adds the required modules if it can find them. Other tools may simply report a dependency or may generate a build failure if all required modules are not included in the project.

Merge modules and their dependencies

Merge Module	BPLs Included	Dependencies
ADO	adortl100.bpl	DatabaseRTL, BaseRTL
BaseRTL	rtl100.bpl	No dependencies
BaseVCL	vcl100.bpl, vclx100.bpl	BaseRTL
BDEInternet	inetdbbde100.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl100.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl100.bpl	BaseRTL
DatabaseVCL	vcldb100.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsnap100.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon100.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapEntera	dsnapent100.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx100.bpl	DatabaseVCL, BaseVCL, BaseRTL, DatabaseRTL
dbExpress	dbexpress100.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds100.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress100.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss100.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
InterbaseVCL	ibxpress100.bpl	BaseClientDataSet, BaseRTL, BaseVCL, DatabaseRTL, DatabaseVCL, DataSnap, dbExpress
Internet	inet100.bpl, inetdb100.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy100.bpl	BaseVCL, BaseRTL
Delphi Office2000Components	dcloffice2k100.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
C++ Office2000Components	bcboffice2k100.bpl	BaseVCL, BaseRTL
Delphi OfficeXPComponents	dclofficexp100.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL

C++ OfficeXPComponents	bcbofficexp100.bpl	BaseVCL, BaseRTL
SOAPRTL	soaprtl100.bpl	BaseRTL, XMLRTL, DatabaseRTL, DataSnap, Internet
TeeChart	tee100.bpl, teedb100.bpl, teeqr100.bpl, teeui100.bpl	BaseVCL, BaseRTL
VCLActionBands	vclactnband100.bpl	BaseVCL, BaseRTL
VCLIE	vclie100.bpl	BaseVCL, BaseRTL
WebDataSnap	webdsnap100.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL
WebSnap	websnap100.bpl, vcljpg100.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl100.bpl	Internet, DatabaseRTL, BaseRTL

ActiveX Controls

Certain components bundled with Delphi are ActiveX controls. The component wrapper is linked into the application's executable file (or a runtime package), but the .ocx file for the component also needs to be deployed with the application. These components include:

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One (spreadsheet), copyright Visual Components, Inc.
- First Impression (VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

ActiveX controls that you create need to be registered on the deployment computer before use. Installation programs such as InstallShield Express automate this registration process. To manually register an ActiveX control, choose **Run** ▸ **ActiveX Server** in the IDE, use the TRegSvr demo application in \Bin or use the Microsoft utility REGSRV32.EXE (not included with Windows 9x versions).

DLLs that support an ActiveX control also need to be distributed with an application.

Helper Applications

Helper applications are separate programs without which your application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by Borland, or by third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the helper program documentation for specific information.

DLL Locations

You can install DLL files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\System

directory. A better way is to create a dedicated directory for the common .DLL files, similar to the way the Borland Database Engine is installed.

Deploying Database Applications

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require additional handling on installation, because the files that make up the application are typically located on multiple computers.

Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are using, you need to make sure that the client-side software is installed on the system where you plan to run the database application. ADO, BDE, dbExpress, and InterBase Express also require drivers to interact with the client-side software of the database.

Specific information on how to deploy dbExpress, BDE, and multi-tiered database applications is described in the following topics:

- Deploying dbExpress Database Applications.
- Deploying BDE Applications.
- Deploying Multi-tiered Database Applications (DataSnap).

Database applications that use client datasets such as *TClientDataSet* or dataset providers require you to include midaslib.dcu (for static linking when providing a stand-alone executable); if you are packaging your application (with the executable and any needed DLLs), you need to include Midas.dll.

If deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 2000 and Internet Explorer version 5 or later. You also need to be sure the drivers for the database server you want to connect to are installed on the client. No other deployment steps are required.

If deploying database applications that use InterBase Express, you need to be sure that the InterBase client is installed on the system where you plan to run the application. InterBase requires gds32.dll and interbase.msg to be located in an accessible directory. No other deployment steps are required. InterBase Express components communicate directly with the InterBase Client API and do not require additional drivers. For more information, refer to the Embedded Installation Guide posted on the Borland Web site.

In addition to the technologies described here, you can also use third-party database engines to provide database access. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

Deploying dbExpress Database Applications

dbExpress is a set of thin, native drivers that provide fast access to database information.

You can deploy dbExpress applications either as a stand-alone executable file or as an executable file that includes associated dbExpress driver DLLs.

To deploy dbExpress applications as stand-alone executable files, the dbExpress object files must be statically linked into your executable. You do this by including the following DCUs, located in the lib directory:

dbExpress deployment as stand-alone executable

Database Unit	When to Include
dbExpINT	Applications connecting to InterBase databases
dbExpORA	Applications connecting to Oracle databases

dbExpDB2	Applications connecting to DB2 databases
dbExpMYS	Applications connecting to MySQL 4.0.24 databases
MidasLib	Required by dbExpress executables that use client datasets such as *TClientDataSet*

Note: For database applications using Informix, you cannot deploy a standalone executable. Instead, deploy an executable file with the driver DLL (listed in the following table).

If you are not deploying a stand-alone executable, you can deploy associated dbExpress drivers and DataSnap DLLs with your executable. The following table lists the appropriate DLLs and when to include them:

dbExpress deployment with driver DLLs

Database DLL	When to Deploy
dbxinf30.dll	Applications connecting to Informix databases
dbxint30.dll	Applications connecting to InterBase databases
dbxora30.dll	Applications connecting to Oracle databases
dbxdb230.dll.	Applications connecting to DB2 databases
dbxmss30.dll	Applications connecting to MSSQL databases
dbxmys30.dll	Applications connecting to MySQL 4.0.24 databases
Midas.dll	Required by database applications that use client datasets

See Using Unidirectional Datasets for more information about using the dbExpress components.

Deploying BDE Applications

The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables.

Database access for an application is provided by various database engines. An application can use the BDE or a third-party database engine. Borland Database Engine describes installation of the database access elements of an application.

Borland Database Engine

You can use the Borland Database Engine (BDE) to provide database access for standard Delphi data components. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE.

You should use InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express creates the necessary registry entries and defines any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

- Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only Borland products, but many third-party programs that use the BDE.
- Under 32-bit Windows 95/NT and later, BDE configuration information is stored in the Windows registry instead of .ini files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallShield Express, are

capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

Deploying Multi-tiered Database Applications (DataSnap)

DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

Install DataSnap along with a multi-tier application using InstallShield Express (or other Borland-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.

Deploying Web Applications

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms.

The steps for deploying Web applications are the same as those for general applications, except the application's files are deployed on the Web server.

Here are some special considerations for deploying Web applications:

- For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server.
- For dbExpress applications, the dbExpress DLLs must be included in the path. If included, the *dbExpress* driver is installed with the application files on the Web server.
- Security for the directories should be set so that the application can access all needed database files.
- The directory containing an application must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.
- The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.

For information on deploying database Web applications, see Deploying database applications.

For information on deploying applications on Apache servers, see Deploying on Apache servers.

Deploying On Apache Servers

WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications.

Modules and applications are enabled and configured by modifying Apache's httpd.conf file (normally located in your Apache installation's \conf directory).

Enabling modules

Your DLLs should be physically located in the Apache Modules subdirectory.

Two modifications to httpd.conf are required to enable a module.

The first modification is to add a LoadModule entry to let Apache locate and load your DLL. For example:

```
LoadModule MyApache_module modules/Project1.dll
```

Replace MyApache_module with the exported module name from your DLL. To find the module name, in your project source, look for the exports line. For example:

```
exports
  apache_module name 'MyApache_module';
```

The second modification is to add a resource locator entry (may be added anywhere in httpd.conf after the LoadModule entry). For example:

```
# Sample location specification for a project named project1.
<Location /project1>
SetHandler project1-handler
</Location>
```

This allows all requests to http://www.somedomain.com/project1 to be passed on to the Apache module.

The SetHandler directive specifies the Web server application that handles the request. The SetHandler argument should be set to the value of the ContentType global variable.

CGI applications

When creating CGI applications, the physical directory (specified in the Directory directive of the httpd.conf file) must have the ExecCGI option and the SetHandler clause set to allow execution of programs so the CGI script can be executed. To ensure that permissions are set up properly, use the Alias directive with both Options ExecCGI and SetHandler enabled.

Note: An alternative approach is to use the ScriptAlias directive (without Options ExecCGI), but using this approach can prevent your CGI application from reading any files in the ScriptAlias directory.

The following httpd.conf line is an example of using the Alias directive to create a virtual directory on your server and mark the exact location of your CGI script:

```
Alias/MyWeb/"c:/httpd/docs/MyWeb/"
```

This would allow requests such as /MyWeb/mycgi.exe to be satisfied by running the script c:\httpd\docs\MyWeb \mycgi.exe.

You can also set Options to All or to ExecCGI using the Directory directive in httpd.conf. The Options directive controls which server features are available in a particular directory.

Directory directives are used to enclose a set of directives that apply to the named directory and its subdirectories. An example of the Directory directive is shown below:

```
<Directory "c:/httpd/docs/MyWeb">
AllowOverride None
Options ExecCGI
Order allow,deny
Allow from all
AddHandler cgi-script exe cgi
</Directory>
```

In this example, Options is set to ExecCGI permitting execution of CGI scripts in the directory MyWeb. The AddHandler clause lets Apache know that files with extensions such as exe and cgi are CGI scripts (executables).

Note: Apache executes locally on the server within the account specified in the User directive in the httpd.conf file. Make sure that the user has the appropriate rights to access the resources needed by the application.

See the Apache LICENSE file, included with your Apache distribution, for additional deployment information. For additional Apache configuration information, see http://www.apache.org.

Programming for Varying Host Environments

Due to the nature of various operating system environments, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Operating system versions
- Helper applications
- DLL locations

Screen Resolutions and Color Depths

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

The following topics are discussed in this section:

- Considerations When Not Dynamically Resizing
- Considerations When Dynamically Resizing Forms and Controls
- Accommodating Varying Color Depths

Considerations When Not Dynamically Resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

Considerations When Dynamically Resizing Forms and Controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.*Height or *TScreen*.Width property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.
- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm*.*Scaled* form's Scaled property to *True* and calling *TWinControl.ScaleBy* its ScaleBy method . The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *TWinControl*.*ScaleBy* method , by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.
- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the AutoSize property of these controls to *False*.
- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

Accommodating Varying Color Depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

Fonts

Windows comes with a standard set of TrueType and raster fonts. Linux comes with a standard set of fonts, depending on the distribution. When designing an application to be deployed on other computers, realize that not all computers have fonts outside the standard sets.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to a Windows application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .fot file for the nonstandard font with the application.

Operating System Versions

When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions.

To account for this possibility, you have a few options:

- Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions.
- Check the version of the operating system as the application is installed. If an incompatible version of the operating system is present, either halt the installation process or at least warn the installer of the problem.
- Check the operating system version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of the operating system is present, abort the process and alert the user. Alternately, provide different code to run dependent on different operating system versions.

Note: Some operations are performed differently on Windows 95/98 than on Windows NT/2000/XP. Use the Windows API function *GetVersionEx* to determine the Windows version.

Software License Requirements

The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

DEPLOY

The DEPLOY document covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Delphi application. The DEPLOY document is installed in the main Delphi directory. The topics covered include:

- .exe, .dll, and .bpl files
- Components and design-time packages
- Borland Database Engine (BDE)

- ActiveX controls
- Sample images

README

The README document contains last minute information about Delphi, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. The README document is installed in the main Delphi directory.

No-nonsense license agreement

The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.

Third-party product documentation

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Delphi applications prior to distribution.

Developing Database Applications

Designing database applications

Designing Database Applications: Overview

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

The following topics introduce common considerations when designing a database application:

- Using Databases
- Database Architecture
- Designing the User Interface

Using Databases

Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism:

- The BDE page of the Component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see Using the Borland Database Engine.
- The ADO page of the Component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets you integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see Working with ADO Components
- The dbExpress page of the Component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see Using unidirectional datasets

- The InterBase page of the Component palette contains components that access InterBase databases directly, without going through a separate engine layer.
- The Data Access page of the Component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups. For more information about using client datasets, see Using client datasets For more information about *TDataSetProvider*, see Using provider components

Note: Different versions of Delphi include different drivers for accessing database servers using the BDE, ADO, or dbExpress.

When designing a database application, you must decide which set of components to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

In addition to choosing a data access mechanism, you must choose a database server. There are different types of databases and you will want to consider the advantages and disadvantages of each type before settling on a particular database server.

All types of databases contain tables which store information. In addition, most (but not all) servers support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

Types of Databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server:

- **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique "dialect" of SQL. Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.
- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered** applications because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some

local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.

- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

Database Security

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see Controlling server login.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If you require your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password at the point when you will eventually log in to the SQL database, rather than when opening individual tables.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password that is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use HTTPs or COM + to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.

- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is not part of most local databases, although it is provided by local InterBase. In addition, the BDE drivers provide limited transaction support for some local databases. Database transaction support is provided by the component that represents the database connection. For details on managing transactions using a database connection component, see Managing transactions.

In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases. For details on using transactions in multi-tiered applications, see Managing transactions in multi-tiered applications.

Referential Integrity, Stored Procedures, and Triggers

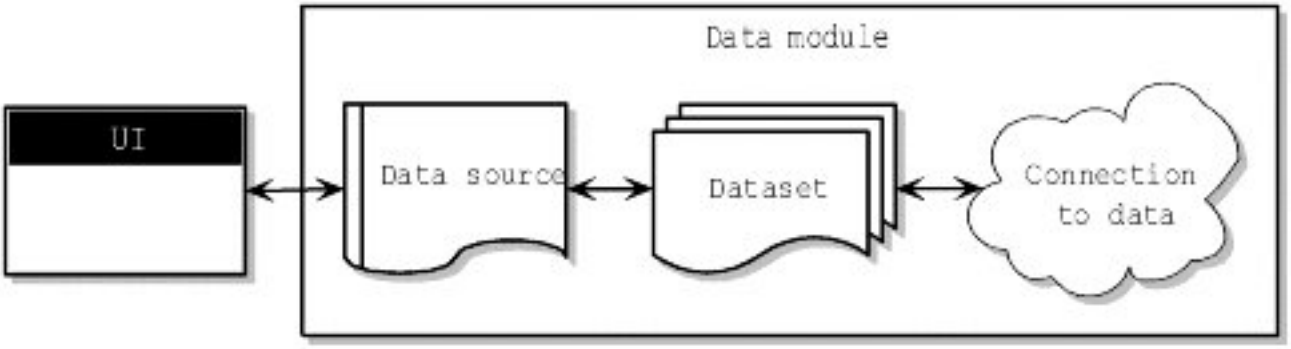
All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

Database Architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in the following figure:



The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see Designing the user interface.

The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Understanding datasets

The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. There are four basic mechanisms for connecting to the data:

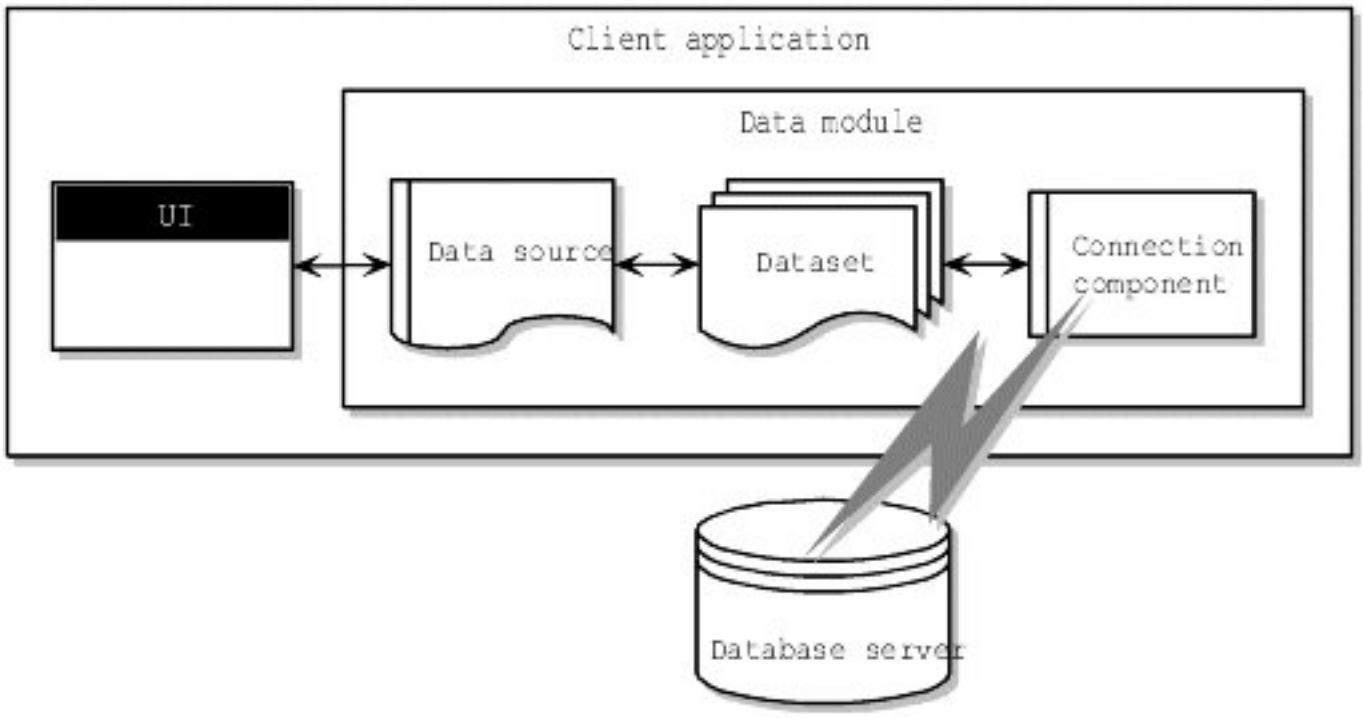
- Connecting directly to a database server. Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk. Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when working with a dedicated file because the client dataset itself knows how to read from and write to the file.
- Connecting to another dataset. Client datasets can work with data provided by another dataset. A *TDataSetProvider* component serves as an intermediary between the client dataset and its source dataset. This dataset provider can reside in the same data module as the client dataset, or it can be part of an application server running on another machine. If the provider is part of an application server, you also need a special descendant of *TCustomConnection* to represent the connection to the application server.
- Obtaining data from an RDS DataSpace object. ADO datasets can use a TRDSConnection component to marshal data in multi-tier database applications that are built using ADO-based application servers.

Sometimes, these mechanisms can be combined in a single application.

Connecting Directly to a Database Server

The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in the following figure.

Connecting directly to the database server



Each type of dataset uses its own type of connection component, which represents a single data access mechanism:

- If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a TDataBaseobject. You connect the dataset to the database component by setting its Databaseproperty. You do not need to explicitly add a database component when using BDE dataset. If you set the dataset's DatabaseName property, a database component is created for you automatically at runtime.
- If the dataset is an ADO dataset such as *TADODataSet*, *TADOTable*, *TADOQuery*, or *TADOStoredProc*, the connection component is a TADOConnectionobject. You connect the dataset to the ADO connection component by setting its Connectionproperty. As with BDE datasets, you do not need to explicitly add the connection component: instead you can set the dataset's ConnectionStringproperty.
- If the dataset is a dbExpress dataset such as *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, or *TSQLStoredProc*, the connection component is a TSQLConnection object. You connect the dataset to the SQL connection component by setting its SQLConnection property. When using dbExpress datasets, you must explicitly add the connection component. Another difference between dbExpress datasets and the other datasets is that dbExpress datasets are always read-only and unidirectional: This means you can only navigate by iterating through the records in order, and you can't use the dataset methods that support editing.
- If the dataset is an InterBase Express dataset such as *TIBDataSet*, *TIBTable*, *TIBQuery*, or *TIBStoredProc*, the connection component is a TIBDatabaseobject. You connect the dataset to the IB database component by setting its Database_Database">Databaseproperty. As with dbExpress datasets, you must explicitly add the connection component.

In addition to the components listed above, you can use a specialized client dataset such as *TBDEClientDataSet, TSimpleDataSet*, or *TIBClientDataSet* with a database connection component. When using one of these client datasets, specify the appropriate type of connection component as the value of the *DBConnection* property.

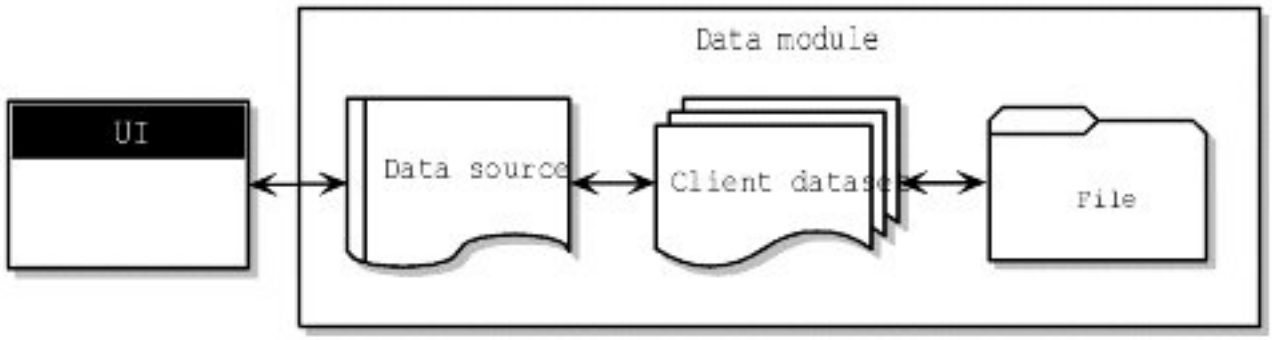
Although each type of dataset uses a different connection component, they all perform many of the same tasks and surface many of the same properties, methods, and events. For more information on the commonalities among the various database connection components, see Connecting to databases

This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

Note: The connection components or drivers needed to create two-tiered applications are not available in all version of Delphi.

Using a Dedicated File on Disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in the following figure:



When using this file-based approach, your application writes changes to disk using the client dataset's SaveToFile method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table.

When you want to read a table previously written using the *SaveToFile* method, use the LoadFromFile method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the FileName property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (If you do not statically link in midaslib.dcu, the client dataset does require midas.dll). There is no need for site licenses or database administration.

In addition, some versions of Delphi let you convert between arbitrary XML documents and the data packets that are used by a client dataset. Thus, the file-based approach can be used to work with XML documents as well as dedicated datasets. For information about converting between XML documents and client dataset data packets, see Using XML in database applications

The file-based approach offers no support for multiple users. The dataset should be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other's data files.

For more information about using a client dataset with data stored on disk, see Using a client dataset with file-based data.

Connecting to Another Dataset

There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits:

- Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using a dataset that caches updates locally and applies them all later in a single transaction. For information on the advantages of using client datasets to cache updates, see Using a client dataset to cache updates.
- Client datasets can apply edits directly to a database server when the dataset is read-only. When using *dbExpress*, this is the only way to edit the data in the dataset (it is also the only way to navigate freely in the data when using *dbExpress*). Even when not using *dbExpress*, the results of some queries and all stored procedures are read-only. Using a client dataset provides a standard way to make such data editable.
- Because client datasets can work directly with dedicated files on disk, using a client dataset can be combined with a file-based model to allow for a flexible "briefcase" application.

In addition to these specialized client datasets, there is a generic client dataset (*TClientDataSet*), which does not include an internal dataset and dataset provider. Although *TClientDataSet* has no built-in database access mechanism, you can connect it to another, external, dataset from which it fetches data and to which it sends updates. Although this approach is a bit more complicated, there are times when it is preferable:

- Because the source dataset and dataset provider are external, you have more control over how they fetch data and apply updates. For example, the provider component surfaces a number of events that are not available when using a specialized client dataset to access data.
- When the source dataset is external, you can link it in a master/detail relationship with another dataset. An external provider automatically converts this arrangement into a single dataset with nested details. When the source dataset is internal, you can't create nested detail sets this way.
- Connecting a client dataset to an external dataset is an architecture that easily scales up to multiple tiers. Because the development process can get more involved and expensive as the number of tiers increases, you

may want to start developing your application as a single-tiered or two-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. If you think you may eventually use a multi-tiered architecture, it can be worthwhile to start by using a client dataset with an external source dataset. This way, when you move the data access and manipulation logic to a middle tier, you protect your development investment because the code can be reused as your application grows.

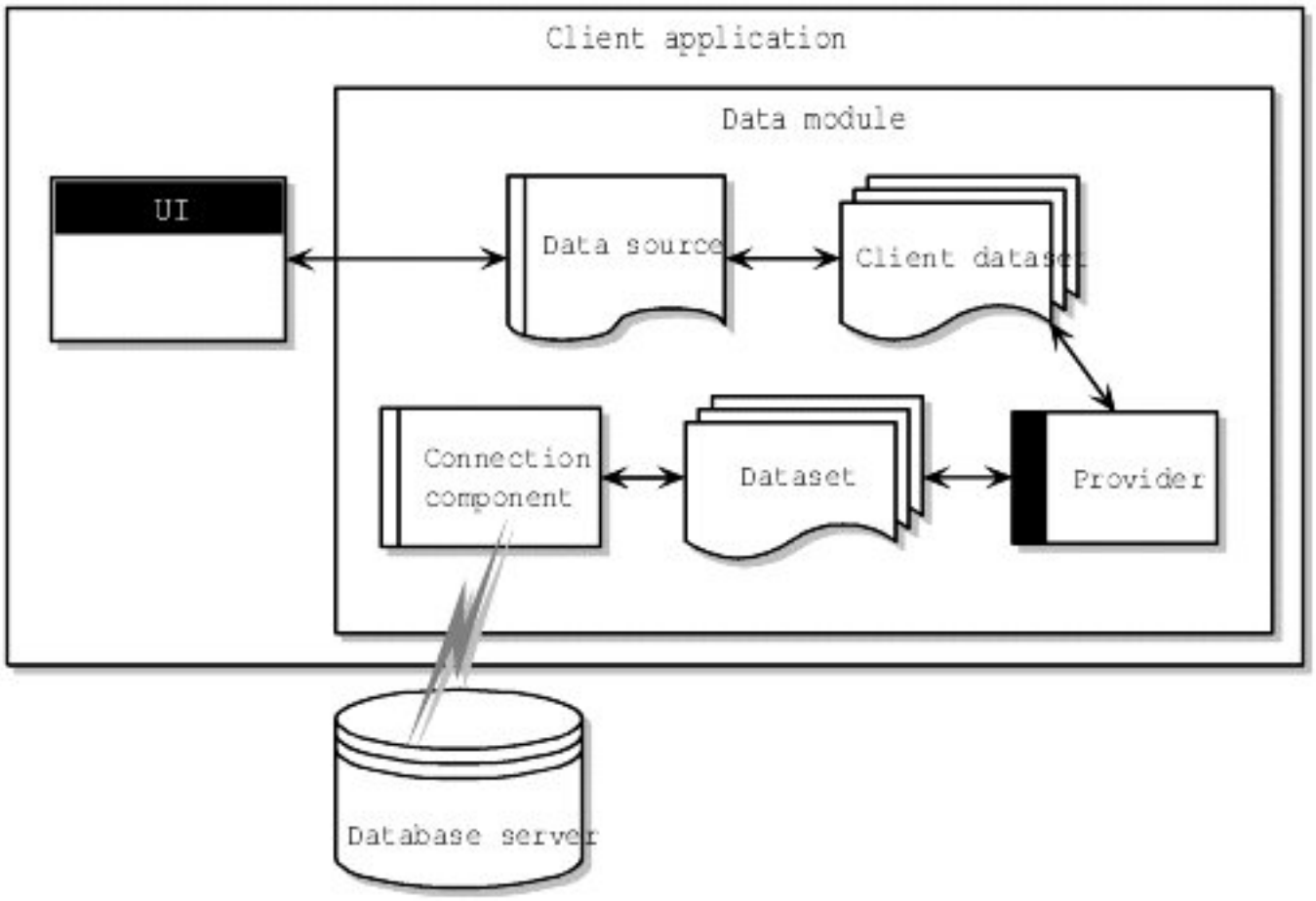
- *TClientDataSet* can link to any source dataset. This means you can use custom datasets (third-party components) for which there is no corresponding specialized client dataset. Some versions of Delphi even include special provider components that connect a client dataset to an XML document rather than another dataset. (This works the same way as connecting a client dataset to another (source) dataset, except that the XML provider uses an XML document rather than a dataset. For information about these XML providers, see Using an XML document as the source for a provider.)

There are two versions of the architecture that connects a client dataset to an external dataset:

- Connecting a client dataset to another dataset in the same application.
- Using a multi-tiered architecture.

Connecting a Client Dataset to Another Dataset in the Same Application

By using a provider component, you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. The architecture for this is illustrated in the following figure.



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

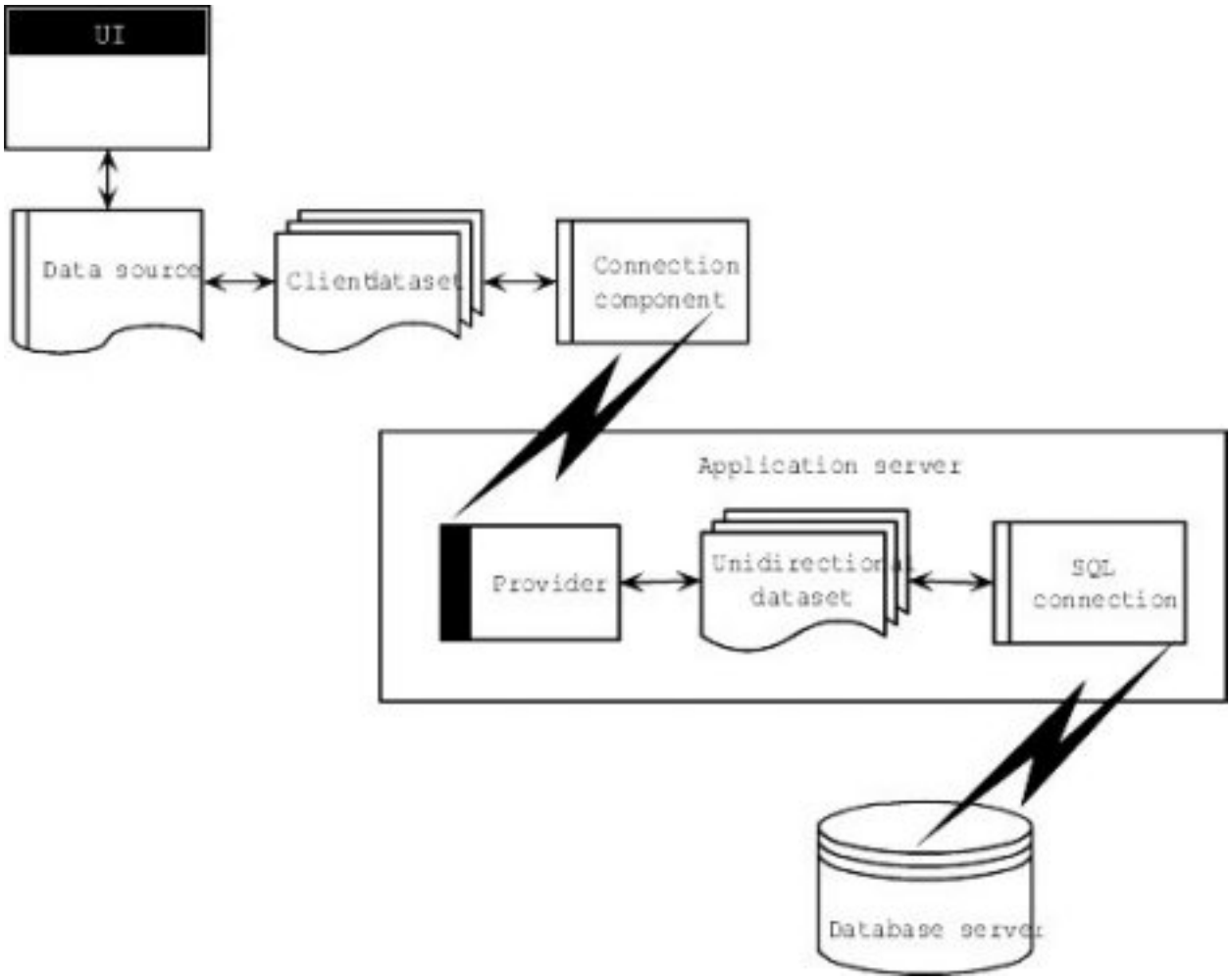
To link the client dataset to the provider, set its ProviderName property to the name of the provider component. The provider must be in the same data module as the client dataset. To link the provider to the source dataset, set its DataSet property.

Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply user edits back to the database, you need only call the client dataset's ApplyUpdates method.

For more information on using a client dataset with a provider, see Using a client dataset with a provider.

Using a Multi-Tiered Architecture

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in the following figure.



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading data-processing over several systems.

The multi-tiered architecture is very similar to the model described in Connecting a client dataset to another dataset in the same application. It differs mainly in that source dataset that connects to the database server and the provider that acts as an intermediary between that source dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the "remote data broker").

Because the provider has moved to a separate application, the client dataset can no longer connect to the source dataset by simply setting its ProviderName property. In addition, it must use some type of connection component to locate and connect to the application server.

There are several types of connection components that can connect a client dataset to an application server. They are all descendants of TCustomRemoteServer, and differ primarily in the communication protocol they use (TCP/IP, HTTP, DCOM, or SOAP). Link the client dataset to its connection component by setting the RemoteServer property.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

For more information about connecting a client dataset to an application server, see Creating multi-tiered applications

Combining Approaches

There is no reason why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in Using a dedicated file on disk with another approach such as those described in Connecting to another dataset or Using a multi-tiered architecture. These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales representatives return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The client dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

Designing the User Interface

The Data Controls category of the **Tool Palette** provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Using data controls.

In addition to the basic data controls, you may also want to introduce other elements into your user interface:

- You may want your application to analyze the data contained in a database. Applications that analyze data do more than just display the data in a database, they also summarize the information in useful formats to help users grasp the impact of that data.
- You may want to print reports that provide a hard copy of the information displayed in your user interface.
- You may want to create a user interface that can be viewed from Web browsers. The simplest Web-based database applications are described in Using database information in responses. In addition, you can combine the Web-based approach with the multi-tiered architecture, as described in Writing Web-based client applications.

Analyzing Data

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The TDBChart component on the Data Controls category of the **Tool Palette** lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube category on the **Tool Palette**. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see Using decision support components

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset.

Writing Reports

If you want to let your users print database information from the datasets in your application, you can use Rave Reports, as described in Creating reports with Rave Reports.

Using data controls

Using Data Controls

The Data Controls category of the **Tool palette** provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in Displaying a Single Record.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. Choosing how to organize the data describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator.

Note: More complex data-aware controls for decision support are discussed in Using Decision Support Components.

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described in Using Common Data Control Features.

Using Common Data Control Features

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display

- Enabling mouse, keyboard, and timer events

Data controls let you display and edit fields of data associated with the current record in a dataset. The following table summarizes the data controls that appear on the Data Controls category of the **Tool palette**.

Data controls

Data control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
TDBText	Displays data from a field as a label.
TDBEdit	Displays data from a field in an edit box.
TDBMemo	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
TDBImage	Displays graphics from a data field in a graphics box.
TDBListBox	Displays a list of items from which to update a field in the current data record.
TDBComboBox	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Displays a check box that indicates the value of a Boolean field.
TDBRadioGroup	Displays a set of mutually exclusive options for a field.
TDBLookupListBox	Displays a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.
TDBCtrlGrid	Displays a configurable, repeating set of data-aware controls within a grid.
TDBRichEdit	Displays formatted data from a field in an edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see Creating Persistent Fields.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

Associating a Data Control with a Dataset

Data controls connect to datasets by using a data source. A data source component (TDataSource) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note: Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset

1. Place a dataset in a data module (or on a form), and set its properties as appropriate.
2. Place a data source in the same data module (or form). Using the **Object Inspector**, set its DataSet property to the dataset you placed in step 1.

3 Place a data control from the Data Access category of the **Tool palette** onto a form.

4 Using the **Object Inspector**, set the *DataSource* property of the control to the data source component you placed in step 2.

5 Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid, TDBCtrlGrid,* and *TDBNavigator* because they access all available fields in the dataset.

6 Set the *Active* property of the dataset to *True* to display data in the control.

For more information about managing the link between the data control and its dataset, see

- Changing the Associated Dataset at Runtime
- Enabling and Disabling the Data Source
- Responding to Changes Mediated by the Data Source

Changing the Associated Dataset at Runtime

In Associating a Data Control with a Dataset, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
[Delphi]
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

```
[C++]
if (CustSource->DataSet == Customers)
  CustSource->DataSet = Orders;
else
  CustSource->DataSet = Customers;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```
[Delphi]
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

```
[C++]
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  DataSource1->DataSet = Form1->Table1;
}
```

Enabling and Disabling the Data Source

The data source has an Enabled property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's DisableControls and EnableControls methods because they affect all attached data sources.

Responding to Changes Mediated by the Data Source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The OnDataChange event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The UpdateData event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after Post is called, but before the data is actually posted to the underlying database server or local cache.

The StateChange event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's State property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```
[Delphi]
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  .
  .
  .
end;
```

```
[C++]
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
  CustTableActivateBtn->Enabled = (CustTable->State == dsInactive);
  CustTableEditBtn->Enabled = (CustTable->State == dsBrowse);
  CustTableCancelBtn->Enabled = (CustTable->State == dsInsert ||
                                 CustTable->State == dsEdit ||
                                 CustTable->State == dsSetKey);
  .
  .
  .
}
```

Note: For more information about dataset states, see Determining Dataset States.

Editing and Updating Data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

Note: Unidirectional datasets never permit users to edit and update data.

The following topics describe how to allow users to edit data using data controls:

- Enabling Editing in Controls On User Entry
- Editing Data in a Control

Enabling Editing in Controls On User Entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's AutoEdit property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a TDBNavigator control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see Navigating and manipulating records. Alternately, you can write code that calls the dataset's Edit method when you want to put the dataset into edit mode.

Editing Data in a Control

A data control can only post edits to its associated dataset if the dataset's CanModify property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

Note: Whether a dataset can update data depends on whether the underlying database table permits updates.

Even if the dataset's *CanModify* property is *True*, the Enabled property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is *True* when the dataset's *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except TDBGrid, when you modify a field, the modification is copied to the underlying dataset when you Tab from the control. If you press Esc before you Tab from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In TDBGrid, modifications are posted when you move to a different record; you can press Esc in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware controls associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

Note: If your application caches updates (for example, using a client dataset), all modifications are posted to an internal cache. These modifications are not applied to the underlying database table until you call the datas et's *ApplyUpdates* method.

Disabling and Enabling Data Display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's EnableControls method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
[Delphi]
CustTable.DisableControls;
try
CustTable.First; { Go to first record, which sets EOF False }
while not CustTable.EOF do { Cycle until EOF is True }
begin
{ Process each record here }
    .
    .
    .
CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
end;
finally
CustTable.EnableControls;
end;
```

```
[C++]
CustTable->DisableControls();
try
{
  // cycle through all records of the dataset
  for (CustTable->First(); !CustTable->EOF; CustTable->Next())
  {
  // Process each record here
  .
  .
  .
  }
}
__finally
{
  CustTable->EnableControls();
}
```

Refreshing Data Display

The Refresh method for a dataset flushes local buffers and re-fetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Enabling Mouse, Keyboard, and Timer Events

The Enabled property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Choosing How to Organize the Data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The TDBNavigator control provides built-in support for many of the functions you may want to perform.

Displaying a Single Record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls category of the **Tool palette** provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the **Tool palette**. For example, the TDBEdit control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field. The following topics describe these components in more detail:

- Displaying Data as Labels
- Displaying and Editing Fields in an Edit Box
- Displaying and Editing Text in a Memo Control
- Displaying and Editing Text in a Rich Edit Memo Control
- Displaying and Editing Graphics Fields in an Image Control
- Displaying and Editing Data in List and Combo Boxes
- Handling Boolean Field Values with Check Boxes
- Restricting Field Values with Radio Controls

Displaying Data as Labels

TDBText is a read-only control similar to the TLabel component on the Standard category of the **Tool palette**. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user

enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

Note: When you place a *TDBText* component on a form, make sure its AutoSize property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

Displaying and Editing Fields in an Edit Box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open TClientDataSet called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

Displaying and Editing Text in a Memo Control

TDBMemo is a data-aware component—similar to the standard TMemo component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the ReadOnly property of the memo control to *True*. To display tabs and permit users to enter them in a memo, set the WantTabs property to *True*. To limit the number of characters users can enter into the database memo, use the MaxLength property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the ScrollBars property. To prevent word wrap, set the WordWrap property to *False*. The Alignment property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the Font property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an AutoDisplay property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and Editing Text in a Rich Edit Memo Control

TDBRichEdit is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well.

Note: While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities.

By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the ReadOnly property of the rich edit control to *True*. To display tabs and permit users to enter them in a memo, set the WantTabs property to *True*. To limit the number of characters users can enter into the database memo, use the MaxLength property. The default value for MaxLength is 0, meaning that there is no character limit other than that imposed by the operating system.

Because the *TDBRichEdit* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBRichEdit* has an AutoDisplay property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBRichEdit* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and Editing Graphics Fields in an Image Control

TDBImage is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the Stretch property to *True* to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an AutoDisplay property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to *False*, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and Editing Data in List and Combo Boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list and combo box controls:

- TDBListBox, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- TDBComboBox, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- TDBLookupListBox, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- TDBLookupComboBox, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

The following topics describe these components in more detail:

- Using TDBListBox and TDBComboBox

- Displaying and Editing Data in Lookup List and Combo Boxes

Note: At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. Backspace and Esc cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the Items property in the **Object Inspector**. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its DataField property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

For *TDBListBox*, the Height property determines how many items are visible in the list box at one time. The IntegralHeight property controls how the last item can appear. If *IntegralHeight* is *False* (the default), the bottom of the list box is determined by the ItemHeight property, and the bottom item may not be completely displayed. If IntegralHeight is *True*, the visible bottom item in the list box is fully displayed.

For *TDBComboBox*, the Style property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the Items list is displayed at runtime:

- *Style* determines the display style of the component:
- *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
- *csSimple*: Combines an edit control with a fixed size list of items that is always displayed. When setting *Style* to *csSimple*, be sure to increase the *Height* property so that the list is displayed.
- *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
- *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.
- DropDownCount: the maximum number of items displayed in the list. If the number of Items is greater than *DropDownCount*, the user can scroll the list. If the number of Items is less than *DropDownCount*, the list will be just large enough to display all the Items.
- ItemHeight: The height of each item when style is *csOwnerDrawFixed*.
- Sorted: If *True*, then the *Items* list is displayed in alphabetical order.

Displaying and Editing Data in Lookup List and Combo Boxes

Lookup list boxes and lookup combo boxes (TDBLookupListBox and TDBLookupComboBox) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains

additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

A lookup field defined for a dataset. To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field.

To specify the lookup field for the list box items

1. Set the DataSource property of the list box to the data source for the dataset containing the lookup field to use.
2. Choose the lookup field to use from the drop-down list for the DataField property.
3. When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

A secondary data source, data field, and key. If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item.

To specify a secondary data source for list box items

1. Set the DataSource property of the list box to the data source for the control.
2. Choose a field into which to insert looked-up values from the drop-down list for the DataFieldproperty. The field you choose cannot be a lookup field.
3. Set the ListSource property of the list box to the data source for the dataset that contain the field whose values you want to look up.
4. Choose a field to use as a lookup key from the drop-down list for the KeyField property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
5. Choose a field whose values to return from the drop-down list for the ListField property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the RowCount property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the DropDownRows property instead.

Note: You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see Defining a lookup list column.

Handling Boolean Field Values with Check Boxes

TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of ValueChecked and ValueUnchecked properties. If the field value matches the

ValueChecked property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

Note: The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to "true," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
[Delphi]
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

```
[C++]
DBCheckBox1->ValueChecked = "true;Yes;On";
```

If the field for the current record contains values of "True," "Yes," or "On," then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to "false," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

Restricting Field Values with Radio Controls

TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The Items property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button's label.

If the current value of a field associated with a radio group matches one of the strings in the Items property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for *Items*, and the field for the current record contains the value "Blue," then the third button in the group appears selected.

Note: If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the Values property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

Displaying Multiple Records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in Viewing and editing data with TDBGrid and Creating a grid that contains other data-aware controls.

Note: You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- **Master-detail forms**: You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see Creating Master/detail Relationships and Establishing master/detail relationships using parameters.
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

Tip: It is generally not a good idea to combine these two approaches on a single form. It is usually confusing for users to understand the data relationships in such forms.

Viewing and Editing Data with TDBGrid

A TDBGrid control lets you view and edit records in a dataset in a tabular grid format.

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see Creating a customized grid.
- Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Working with field components.
- The dataset's *ObjectView* property setting for grids displaying ADT and array fields. See Displaying ADT and array fields.

A grid control has a Columns property that is itself a wrapper on a *TDBGridColumns* object. TDBGridColumns is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

The following topics describe how to use the TDBGrid component:

- Using a Grid Control in Its Default State
- Creating a Customized Grid

- Displaying ADT and Array Fields
- Setting Grid Options
- Editing in the Grid
- Controlling Grid Drawing
- Responding to User Actions at Runtime

Using a Grid Control in Its Default State

The Stateproperty of the grid's *Columns* property indicates whether persistent column objects exist for the grid. *Columns.State* is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid's *Columns.State* property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's *DataSource* property changes or when the *DataSet* property of the data source itself is changed.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the Width property of a column changes the DisplayWidth property of the field associated with that column. Changes made to column properties that are not based on field properties, such as Font, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Note: Changing a grid's *Columns.State* property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

Creating a Customized Grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's DisplayLabel property. If you modify the *DisplayLabel* property, the column title reflects that change immediately. If you then assign a string to the column title's caption, the tile caption becomes independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's FieldName property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's Field property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of the record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

Note: Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's FieldCount property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's SelectedField property is NULL and the SelectedIndexproperty is –1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (...) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

The following topics provide additional information about persistent columns:

- Creating Persistent Columns
- Deleting Persistent Columns
- Arranging the Order of Persistent Columns
- Setting Column Properties at Design Time
- Defining a Lookup List Column
- Putting a Button in a Column
- Restoring Default Values to a Column

Creating Persistent Columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the State property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control

1 Select the grid component in the form.

2 Invoke the Columns editor by double clicking on the grid's Columns property in the **Object Inspector**.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis.

To create persistent columns for all fields

1 Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.
2 If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.
3 Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually

1 Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).
2 To associate a field with this new column, set the FieldName property in the **Object Inspector**.
3 To set the title for the new column, expand the Title property in the **Object Inspector** and set its *Caption* property.
4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
[Delphi]
DBGrid1.Columns.Add;
```

```
[C++]
DBGrid1->Columns->Add();
```

Deleting Persistent Columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display.

To remove a persistent column from a grid

1 Double-click the grid to display the Columns editor.
2 Select the field to remove in the Columns list box.

3 Click Delete (you can also use the context menu or Del key, to remove a column).

Note: If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
[Delphi]
DBGrid1.Columns[5].Free;
```

```
[C++]
delete DBGrid1->Columns->Items[5];
```

Arranging the Order of Persistent Columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column

1 Select the column in the Columns list box.
2 Drag it to a new location in the list box.

You can also change the column order at runtime by clicking on the column title and dragging the column to a new position.

Note: Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Warning: You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its DragMode property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's OnColumnMoved event fires after a column has been moved.

Setting Column Properties at Design Time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component.

To set a column's properties, select the column in The Columns editor and set its properties in the **Object Inspector**. The following table summarizes key column properties you can set.

Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField.Alignment*.

ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. *cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default source: *TDBGrid.Color*. (For text foreground color, see the Font property.)
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing ADT or array fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	*True*: The data in the column cannot be edited by the user. *False*: (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default source: *TField.DisplayWidth*.
Font	Specifies the font type, size, and color used to draw text in the column. Default source: *TDBGrid.Font*.
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the Title property.

Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default source: *TField.DisplayLabel*.
Color	Specifies the background color used to draw the column title cell. Default source: *TDBGrid.FixedColor*.
Font	Specifies the font type, size, and color used to draw text in the column title. Default source: *TDBGrid.TitleFont*.

Defining a Lookup List Column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's ButtonStyle property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. Once the lookup field is defined, set the column's FieldName to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the PickList property for the column in the **Object Inspector**. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the DropDownRows property.

Note: To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

Putting a Button in a Column

A column can display an ellipsis button (...) to the right of the normal cell editor. Ctrl+Enter or a mouse click fires the grid's OnEditButtonClick event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

Restoring Default Values to a Column

At runtime you can test a column's AssignedValues property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's RestoreDefaults method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
[Delphi]
DBGrid1.Columns.RestoreDefaults;
```

```
[C++]
DBGrid1->Columns->RestoreDefaults();
```

Displaying ADT and Array Fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset.
- It can display composite fields in a single column, reflecting the fact that they are a single field.

When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

To display composite fields as if they were flattened out, set the dataset's ObjectView property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expa*nded property of the column:

- When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
- When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse

The following figure shows a grid with an ADT field and an array field. The dataset's *ObjectView* property is set to *False* so that each child field has a column.

ID_KEY	NAME_ADT.FIRST	NAME_ADT.MIDDLE	NAME_ADT.LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARR
1	Stephan		Wright	415-908-9875	902-786-1245	
2	Whitney	N	Long			
3	Chris	T	Scanlan	234-232-1343		

TDBGrid control with *ObjectView* set to *False*

The following figures show the grid with an ADT field and an array field. The first figure shows the fields collapsed. In this state they cannot be edited. The second figure shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

ID_KEY	NAME_ADT	TELNOS_ARRAY
1	(Stephan, , Wright)	(415-908-9875, 902-786-1245, ...)
2	(Whitney, N, Long)	(, , , 510-454-7234, , , , ,)
3	(Chris, T, Scanlan)	(234-232-1343, ...)

TDBGrid control with *Expanded* set to *False*

ID_KEY	NAME_ADT			TELNOS_ARRAY			
	FIRST	MIDDLE	LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARRAY[2]	TELNO
1	Stephan		Wright	415-908-9875	902-786-1245		
2	Whitney	N	Long				510-454
3	Chris	T	Scanlan	234-232-1343			

TDBGrid control with *Expanded* set to *True*

The following table lists the properties that affect the way ADT and array fields appear in a *TDBGrid*:

Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

Note: In addition to ADT and array fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

Setting Grid Options

You can use the grid Options property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the **Object Inspector** is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the **Object Inspector** below the *Options* property. The + sign changes to a –(minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Expanded TDBGrid Options properties

Option	Purpose
dgEditing	*True*: (Default). Enables editing, inserting, and deleting records in the grid. False: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	*True*: When a field is selected, it is in Edit state. False: (Default). A field is not automatically in Edit state when selected.
dgTitles	*True*: (Default). Displays field names across the top of the grid. False: Field name display is turned off.
dgIndicator	*True*: (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. False: The indicator column is turned off.
dgColumnResize	*True*: (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying *TField* component. False: Columns cannot be resized in the grid.
dgColLines	*True*: (Default). Displays vertical dividing lines between columns. False: Does not display dividing lines between columns.

dgRowLines	*True*: (Default). Displays horizontal dividing lines between records. False: Does not display dividing lines between records.
dgTabs	*True*: (Default). Enables tabbing between fields in records. False: Tabbing exits the grid control.
dgRowSelect	*True*: The selection bar spans the entire width of the grid. False: (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	*True*: (Default). The selection bar in the grid is always visible, even if another control has focus. False: The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	*True*: (Default). Prompt for confirmation to delete records (Ctrl+Del). False: Delete records without confirmation.
dgCancelOnExit	*True*: (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. False: Permits pending inserts.
dgMultiSelect	*True*: Allows user to select noncontiguous rows in the grid using Ctrl+Shift or Shift+ arrow keys. False: (Default). Does not allow user to multi-select rows.

Editing in the Grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The CanModify property of the *Dataset* is *True*.
- The ReadOnly property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, the grid raises an exception, and does not modify the record.

Note: If your application caches updates, posting record changes only adds them to an internal cache. They are not posted back to the underlying database table until your application applies the updates.

You can cancel all edits for a record by pressing Esc in any field before moving to another record.

Controlling Grid Drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or EditFormat properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's OnDrawColumnCell event. If the grid's DefaultDrawing property is True, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *False* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* inside your *OnDrawColumnCell* event handler to have the

grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

Responding to User Actions at Runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the **Object Inspector**.

Grid control events

Event	Purpose
OnCellClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.
OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*.
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Occurs when a user releases a key when in the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

There are many uses for these events. For example, you might write a handler for the OnDblClick event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the SelectedField property to determine to current row and column.

Creating a Grid That Contains Other Data-aware Controls

A TDBCtrlGrid control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row.

To use a database control grid

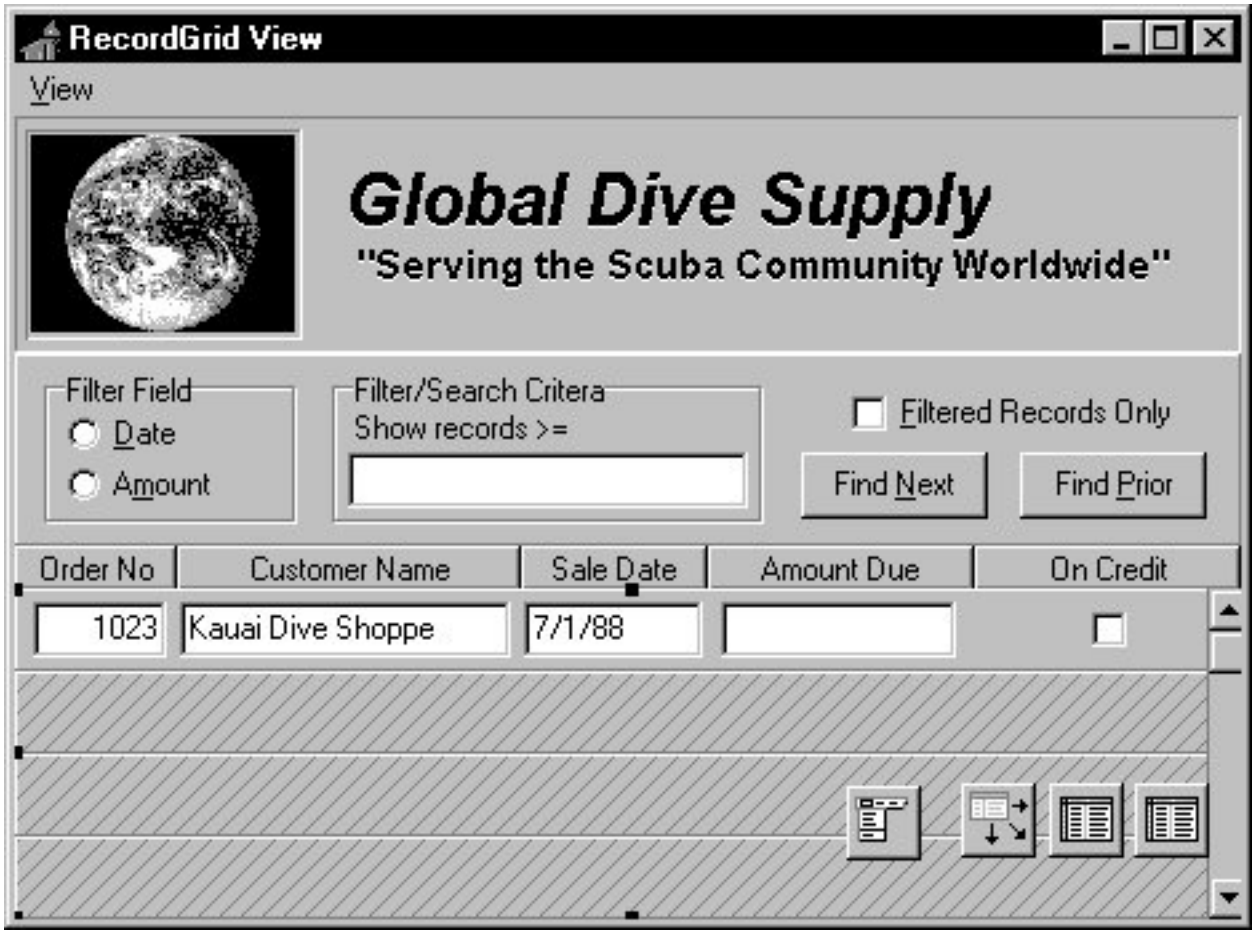
1. Place a database control grid on a form.
2. Set the grid's DataSource property to the name of a data source.

3 Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.

4 Set the *DataField* property for each data control to the name of a field. The data source for these data controls is already set to the data source of the database control grid.

5 Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.



The following table summarizes some of the unique properties for database control grids that you can set at design time:

Selected database control grid properties

Property	Purpose
AllowDelete	*True* (default): Permits record deletion. *False*: Prevents record deletion.
AllowInsert	*True* (default): Permits record insertion. *False*: Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	*goVertical* (default): Display records from top to bottom. *goHorizontal*: Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.

PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	*True* (default): Displays a focus rectangle around the current record's panel at runtime. *False*: Does not display a focus rectangle.

Navigating and Manipulating Records

TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

The following figure shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically. See Choosing Navigator Buttons to Display for more information.

The following table describes the buttons on the navigator.

TDBNavigator buttons

Button	Purpose
First	Calls the dataset's *First* method to set the current record to the first record.
Prior	Calls the dataset's *Prior* method to set the current record to the previous record.
Next	Calls the dataset's *Next* method to set the current record to the next record.
Last	Calls the dataset's *Last* method to set the current record to the last record.
Insert	Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the *ConfirmDelete* property is *True* it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

See Displaying fly-over Help for information on associating help hints with each button. See Using a Single Navigator for Multiple Datasets for information about associating a navigator with multiple datasets.

Choosing Navigator Buttons to Display

When you first place a TDBNavigator on a form at design time, all its buttons are visible. You can use the VisibleButtons property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the *First*, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Hiding and showing navigator buttons at design time

The VisibleButtons property in the **Object Inspector** is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The

list of buttons that can be turned on or off appears in the **Object Inspector** below the *VisibleButtons* property. The + sign changes to a –(minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the TDBNavigator. If *False*, the button is removed from the navigator at design time and runtime.

Note: As button values are set to *False*, they are removed from the TDBNavigator on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The VisibleButtons property controls which buttons are displayed in the navigator. Here's one way you might code the event handler:

```
[Delphi]
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

```
[C++]
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
  {
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbFirst << nbPrior << nbNext << nbLast;
  }
  else
  {
    DBNavigatorAll->DataSource = OrderNum->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbInsert << nbDelete << nbEdit
                                   << nbPost << nbCancel << nbRefresh;
  }
}
```

Displaying Fly-over Help

To display fly-over help for each navigator button at runtime, set the navigator ShowHint property to True. When *ShowHint* is True, the navigator displays fly-by Help hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The Hints property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

Using a Single Navigator for Multiple Datasets

As with other data-aware controls, a navigator's DataSource property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an OnEnter event handler for one of the edit controls, and then share that event with the other edit control. For example:

```
[Delphi]
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

```
[C++]
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
  else
    DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

Creating reports with Rave Reports

Rave Reports: Overview

Rave Reports is a component-based visual report design tool that simplifies the process of adding reports to an application. You can use Rave Reports to create a variety of reports, from simple banded reports to more complex, highly customized reports. Report features include:

- Word wrapped memos
- Full graphics
- Justification
- Precise page positioning
- Printer configuration
- Font control
- Print preview
- Reuse of report content
- PDF, HTML, RTF, and text report renditions

Getting Started with Rave Reports

You can use Rave Reports in VCL applications to generate reports from database and non-database data.

To add a simple report to an existing database application

1. Open a database application in Delphi.
2. From the Rave category of the **Tool palette**, add the TRvDataSetConnection component to a form in the application.
3. In the **Object Inspector**, set the *DataSet* property to a dataset component that is already defined in your application.
4. Use the Rave Visual Designer:
5. From the Rave category of the **Tool palette**, add the Rave project component, TRvProject, to the form.
6. In the **Object Inspector**, set the *ProjectFile* property to the report project file (MyRave.rav) that you created in step 8 in using the Rave Visual Designer.
7. From the Standard category of the **Tool palette**, add the TButton component.

8 In the **Object Inspector**, click the Events tab and double-click the OnClick event.

9 Write an event handler that uses the ExecuteReport method to execute the Rave project component.

To design your report and create a report project file (.rav file) using the Rave Visual Designer

1 Choose **Tools** ▸ **Rave Designer** to launch the Rave Visual Designer.

2 Choose **File** ▸ **New Data Object** to display the Data Connections dialog box, and in the Data Object Type list, select Direct Data View and click Next.

3 In the Active Data Connections list, select RVDataSetConnection1 and click Finish.

In the Project Tree on the left side of the Rave Visual Designer window, expand the Data View Dictionary node, then expand the newly created DataView1 node. Your application data fields are displayed under the DataView1 node.

4 Choose **Tools** ▸ **Report Wizards** ▸ **Simple Table** to display the Simple Table wizard, and select DataView1 and click Next.

5 Select two or three fields that you want to display in the report and click Next.

6 Follow the prompts on the subsequent wizard pages to set the order of the fields, margins, heading text, and fonts to be used in the report.

7 On the final wizard page, click Generate to complete the wizard and display the report in the Page Designer.

8 Choose **File** ▸ **Save** as to display the Save As dialog box. Navigate to the directory in which your Delphi application is located and save the Rave project file as MyRave.rav.

9 Minimize the Rave Visual Designer window and return to Delphi.

For a more information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in Getting more information.

Rave Visual Designer

To launch the Rave Visual Designer, do one of the following:

- Choose **Tools** ▸ **Rave Designer**.
- Double-click a TRvProject component on a form.
- Right-click a TRvProject component on a form, and choose Rave Visual Designer.

For a detailed information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in Getting more information.

Rave Component Overview

This section provides an overview of the Rave Reports components. For detailed component information, see the documentation listed in Getting more information.

VCL components

The VCL components for Rave Reports are non-visual components that you add to a form in your VCL application. They are available on the Rave category of the **Tool palette**. There are four categories of components: engine, render, data connection and Rave project.

Engine components

The Engine components are used to generate reports. Reports can be generated from a pre-defined visual definition (using the *Engine* property of TRvProject) or by making calls to the Rave code-based API library from within the OnPrint event. The engine components are:

TRvNDRWriter

TRvSystem

Render components

The Render components are used to convert an NDR file (Rave snapshot report file) or a stream generated from TRvNDRWriter to a variety of formats. Rendering can be done programmatically or added to the standard setup and preview dialogs of TRvSystem by dropping a render component on an active form or data module within your application. The render components are:

TRvRenderPreview

TRvRenderPrinter

TRvRenderPDF

TRvRenderHTML

TRvRenderRTF

TRvRenderText

Data connection components

The Data Connection components provide the link between application data and the Direct Data Views in visually designed Rave reports. The data connection components are:

TRvCustomConnection

TRvDataSetConnection

TRvTableConnection

TRvQueryConnection

Rave project component

The TRvProject component interfaces with and executes visually designed Rave reports within an application. Normally a TRvSystem component would be assigned to the *Engine* property. The reporting project (.rav) should be specified in the *ProjectFile* property or loaded into the DFM using the *StoreRAV* property. Project parameters can be set using the SetParam method and reports can be executed using the ExecuteReport method.

Reporting components

The following components are available in the Rave Visual Designer.

Project components

The Project toolbar provides the essential building blocks for all reports. The project components are:

TRaveProjectManager

TRaveReport

TRavePage

Data objects

Data objects connect to data or control access to reports from the Rave Reporting Server. The **File ▸ New Data Object** menu command displays the Data Connections dialog box, which you can use to create each of the data objects. The data object components are:

TRaveDatabase
TRaveDriverDataView
TRaveDirectDataView
TRaveSimpleSecurity
TRaveLookupSecurity

Standard components

The Standard toolbar provides components that are frequently used when designing reports. The standard components are:

TRaveText
TRaveMemo
TRaveSection
TRaveBitmap
TRaveMetaFile
TRaveFontMaster
TRavePageNumInit

Drawing components

The Drawing toolbar provides components to create lines and shapes in a report. To color and style the components, use the Fills, Lines, and Colors toolbars. The drawing components are:

TRaveLine
TRaveHLine
TRaveVLine
TRaveSquare
TRaveRectangle
TRaveCircle
TRaveEllipse

Report components

The Report toolbar provides components that are used most often in data-aware reports. The report components are:

TRaveRegion
TRaveDataBand
TRaveBand

Band Style Edito
TRaveDataText
DataText Editor
TRaveDataMemo
TRaveCalcText
TRaveDataCycle
TRaveDataMirrorSection
TRaveCalcOp Component
TRaveCalcController
TRaveCalcTotal

Bar code components

The Bar Code toolbar provides different types of bar codes in a report. The bar code components are:

TRavePostNetBarCode
TRaveI2of5Bar Code
TRaveCode39BarCode
TRaveCode128BarCode
TRaveUPCBarCode
TRaveEANBarCode

Getting More Information

Delphi includes the following Nevrona Designs documentation for Rave Reports.

Rave Reports documentation

Title	Description
Rave Visual Designer Manual for Reference and Learning	Provides detailed information about using the Rave Visual Designer to create reports.
Rave Tutorial and Reference	Provides step-by-step instructions on using the Rave Reports components and includes a reference of classes, components, and units.
Rave Application Interface Technology Specification	Explains how to create custom Rave Reports components, property editors, component editors, project editors, and control the Rave environment.

These books are distributed as PDF files on the Delphi Companion Tools CD.

Most of the information in the PDF files is also available in the online Help. To display online Help for a Rave Reports component on a form, select the component and press F1. To display online Help for the Rave Visual Designer, use the Help menu.

Using decision support components

Using Decision Support Components

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see About crosstabs.

The following topics are discussed in this section:

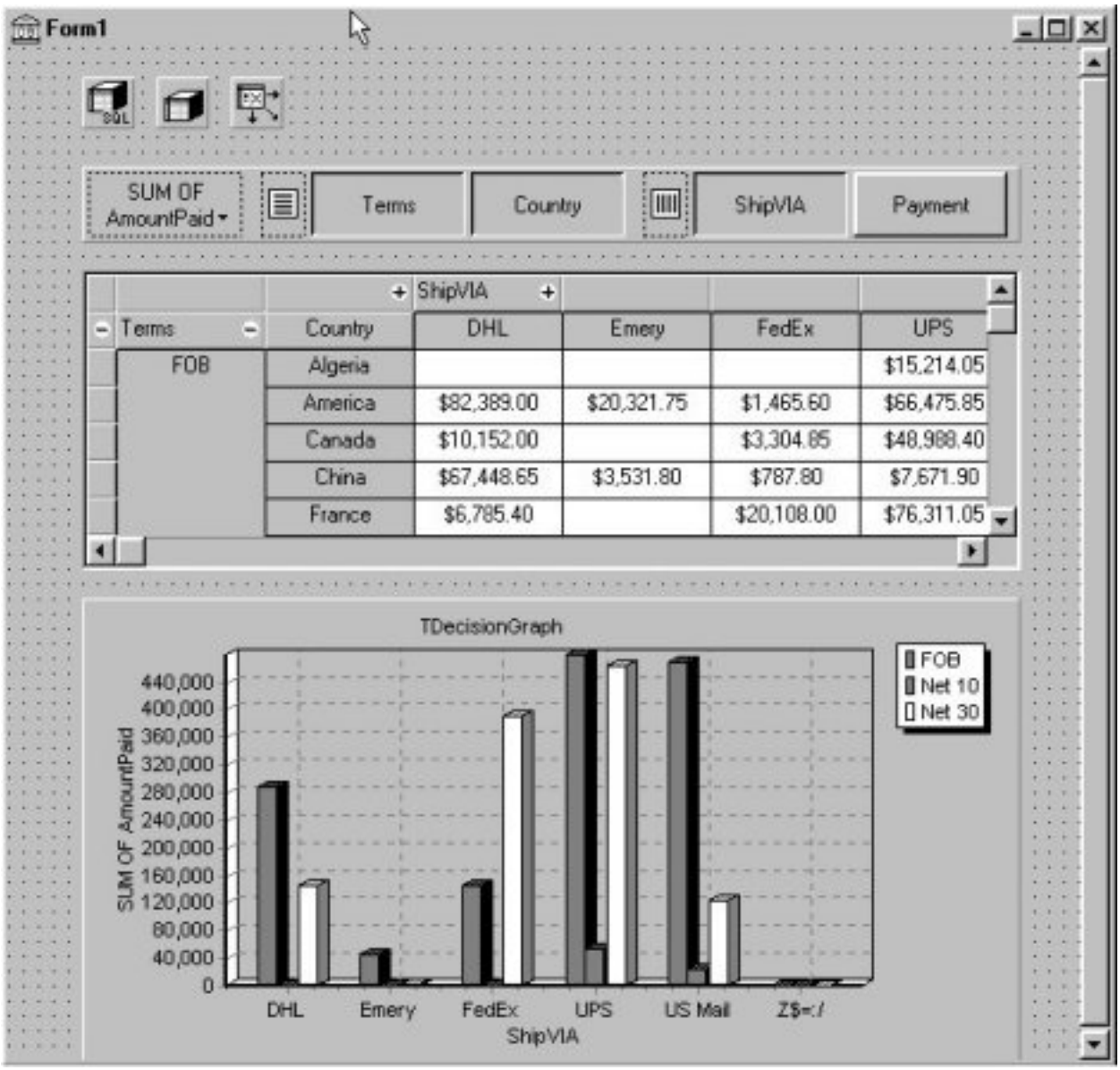
- Overview of Decision Support Components
- Guidelines for Using Decision Support Components
- Decision Support Components at Runtime
- Decision Support Components and Memory Control

Overview of Decision Support Components

The decision support components appear on the Decision Cube category of the **Tool Palette**:

- The decision cube, TDecisionCube, is a multidimensional data store. For more information see Using decision cubes.
- The decision source, TDecisionSource, defines the current pivot state of a decision grid or a decision graph. For more information, see Using decision sources.
- The decision query, TDecisionQuery, is a specialized form of TQuery used to define the data in a decision cube. For more information, see Using datasets with decision support components.
- The decision pivot, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. For more information, see Using decision pivots.
- The decision grid, TDecisionGrid, displays single- and multidimensional data in table form. For more information, see Creating and using decision grids.
- The decision graph, TDecisionGraph, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified. For more information, see Creating and using decision graphs.

The following figure shows all the decision support components placed on a form at design time.



About Crosstabs

Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension.

You can use the decision support components to set up crosstabs in forms. TDecisionGrid shows data in a table, while TDecisionGraph charts it graphically. TDecisionPivot has buttons that make it easier to display and hide dimensions and move them between columns and rows.

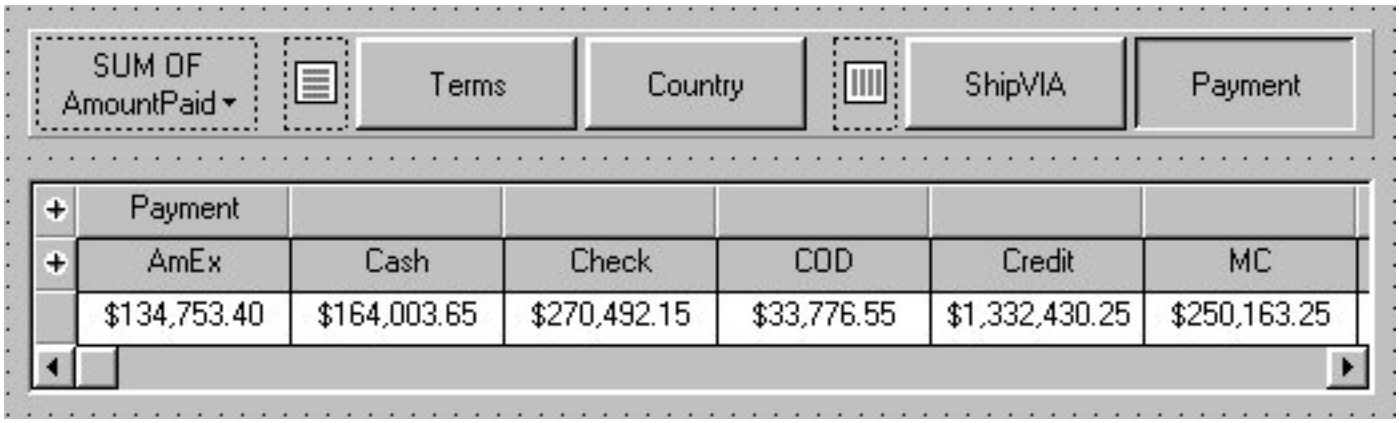
Crosstabs can be one-dimensional or multidimensional.

The following topics are discussed in this section:

- One-Dimensional Crosstabs
- Multidimensional Crosstabs

One-Dimensional Crosstabs

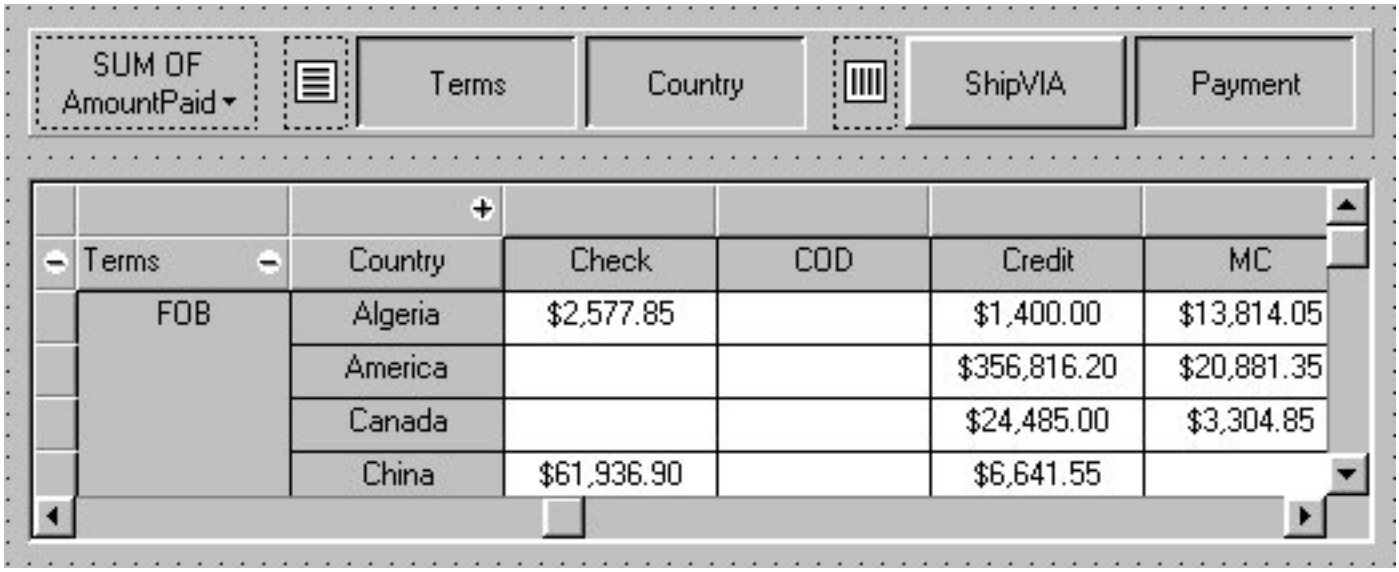
One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if Payment is the chosen column dimension and Amount Paid is the summary category, the crosstab in the following figure shows the amount paid using each method.



Multidimensional Crosstabs

Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country.

A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in the following figure.



Guidelines for Using Decision Support Components

The decision support components listed in Overview of decision support components can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

To create a form with tables and graphs of multidimensional data

1. Create a form.
2. Add these components to the form and use the **Object Inspector** to bind them as indicated:
   - A dataset, usually *TDecisionQuery* (for details, see Creating Decision Datasets with The Decision Query Editor) or TQuery
   - A decision cube, *TDecisionCube*, bound to the dataset by setting its DataSet property to the dataset's name
   - A decision source, *TDecisionSource*, bound to the decision cube by setting its *DecisionCube* property to the decision cube's name
3. Add a decision pivot, *TDecisionPivot*, and bind it to the decision source with the **Object Inspector** by setting its *DecisionSource* property to the appropriate decision source name. The decision pivot is optional but useful; it lets the form developer and end users change the dimensions displayed in decision grids or decision graphs by pushing buttons.

   In its default orientation, horizontal, buttons on the left side of the decision pivot apply to fields on the left side of the decision grid (rows); buttons on the right side apply to fields at the top of the decision grid (columns).

   You can determine where the decision pivot's buttons appear by setting its GroupLayout property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default). For more information on decision pivot properties, see Using decision pivots.
4. Add one or more decision grids and graphs, bound to the decision source. For details, see Creating and using decision grids and Creating and using decision graphs.
5. Use the Decision Query editor or SQL property of *TDecisionQuery* (or *TQuery*) to specify the tables, fields, and summaries to display in the grid or graph. The last field of the SQL SELECT should be the summary field. The other fields in the SELECT must be GROUP BY fields. For instructions, see Creating decision datasets with the Decision Query editor.
6. Set the *Active* property of the decision query (or alternate dataset component) to *True*.
7. Use the decision grid and graph to show and chart different data dimensions. See Using decision grids and Using decision graphs. for instructions and suggestions

For an illustration of all decision support components on a form, see the figure Decision support components at design time.

Using Datasets with Decision Support Components

The only decision support component that binds directly to a dataset is the decision cube, TDecisionCube. *TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified.

The decision query component, TDecisionQuery, is a specialized form of *TQuery*. You can use TDecisionQuery to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube*). The decision query has no properties than are not inherited from other components. Important inherited properties are Active and SQL.

You can also use a TQuery or TTable component as a dataset for *TDecisionCube*, but the correct setup of the dataset and TDecisionCube are then the responsibility of the designer.

To work correctly with the decision cube, all projected fields in the dataset must either be dimensions or summaries. The summaries should be additive values (like sum or count), and should represent totals for each combination of dimension values. For maximum ease of setup, sums should be named "Sum..." in the dataset while counts should be named "Count...".

The Decision Cube can pivot, subtotal, and drill-in correctly only for summaries whose cells are additive. (SUM and COUNT are additive, while AVERAGE, MAX, and MIN are not.) Build pivoting crosstab displays only for grids that contain only additive aggregators. If you are using non-additive aggregators, use a static decision grid that does not pivot, drill, or subtotal.

Since averages can be calculated using SUM divided by COUNT, a pivoting average is added automatically when SUM and COUNT dimensions for a field are included in the dataset. Use this type of average in preference to an average calculated using an AVERAGE statement.

Averages can also be calculated using COUNT(*). To use COUNT(*) to calculate averages, include a "COUNT(*) COUNTALL" selector in the query. If you use COUNT(*) to calculate averages, the single aggregator can be used for all fields. Use COUNT(*) only in cases where none of the fields being summarized include blank values, or where a COUNT aggregator is not available for every field.

Creating Decision Datasets with TQuery or TTable

If you use an ordinary TQuery component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase.

The SQL should look similar to this:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

The ordering of the SELECT fields should match the ordering of the GROUP BY fields. Queries are described in more detail in Using TQuery.

With TTable, you must supply information to the decision cube about which of the fields in the query are grouping fields, and which are summaries. To do this, Fill in the Dimension Type for each field in the *DimensionMap* of the Decision Cube. You must indicate whether each field is a dimension or a summary, and if a summary, you must provide the summary type. Since pivoting averages depend on SUM/COUNT calculations, you must also provide the base field name to allow the decision cube to match pairs of SUM and COUNT aggregators.

Creating Decision Datasets with the Decision Query Editor

All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See Using datasets with decision support components for more information.

While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use TDecisionQuery; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.

To use the Decision Query editor

1. Select the decision query component on the form, then right-click and choose Decision Query editor. The Decision Query editor dialog box appears.
2. Choose the database to use.
3. For single-table queries, click the Select Table button.

   For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.
4. Return to the Decision Query editor dialog box.

5 In the Decision Query editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.

6 By default, all fields and summaries defined in the SQL property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see Using decision cubes,Using decision sources, and Using decision pivots.

Note: When you use the Decision Query editor, the query is initially handled in ANSI-92 SQL syntax, then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The dialect translation is automatically assigned to the *TDecisionQuery*'s SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather then the SQL property.

Using Decision Cubes

The decision cube component, TDecisionCube, is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time.

The following topics are discussed in this section:

- Decision Cube Properties and Events
- Using the Decision Cube Editor

Decision Cube Properties and Events

The DimensionMap properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source.

When you click the ellipsis next to *DimensionMap* in the **Object Inspector**, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.

The OnRefresh event fires whenever the decision cube cache is rebuilt. Developers can access the new dimension map and change it at that time to free up memory, change the maximum summaries or dimensions, and so on. *OnRefresh* is also useful if users access the Decision Cube editor; application code can respond to user changes at that time.

Using the Decision Cube Editor

You can use the Decision Cube editor to set the DimensionMap properties of decision cubes. You can display the Decision Cube editor through the **Object Inspector**, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor.

The Decision Cube Editor dialog box has two tabs:

- Dimension Settings, used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display.

- Memory Control, used to set the maximum number of dimensions and summaries that can be active at one time, to display information about memory usage, and to determine the names and data that appear at design time.

Viewing and Changing Dimension Settings

To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor:

- To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box.
- To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify whether the selected field is a dimension or summary.
- To disable or activate the selected dimension or summary, change the setting in the Active Type drop-down list box: Active, As Needed, or Inactive. Disabling a dimension or setting it to As Needed saves memory.
- To change the format of that dimension or summary, enter a format string in the Format edit box.
- To display that dimension or summary by Year, Quarter, or Month, change the setting in the Binning drop-down list box. Note that you can choose Set in the Binning list box to put the selected dimension or summary in a permanently "drilled down" state. This can be useful for saving memory when a dimension has many values. For more information, see Decision support components and memory control.
- To determine the starting value for ranges, or the drill-down value for a "Set" dimension, first choose the appropriate Grouping value in the Grouping drop-down, and then enter the starting range value or permanent drill-down value in the Initial Value drop-down list.

Setting the Maximum Available Dimensions and Summaries

To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see Decision Support Components and Memory Control.

Viewing and Changing Design Options

To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.

Using Decision Sources

The decision source component, TDecisionSource, defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states.

The following are some special properties and events that control the appearance and behavior of decision sources:

- The ControlType property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).

- The SparseCols and SparseRows properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.

TDecisionSource has the following events:

- OnLayoutChange occurs when the user performs pivots or drill-downs that reorganize the data.
- OnNewDimensions occurs when the data is completely altered, such as when the summary or dimension fields are altered.
- OnSummaryChange occurs when the current summary is changed.
- OnStateChange occurs when the Decision Cube activates or deactivates.
- OnBeforePivot occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
- OnAfterPivot fires after a change in pivot state. Developers can capture information at that time.

Using Decision Pivots

The decision pivot component, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the TDecisionGrid or TDecisionGraph component. When a dimension is closed, its detailed data doesn't appear; it collapses into the totals of other dimensions. A dimension may also be in a "drilled" state, where only the summaries for a particular value of the dimension field appear.

You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag a button to the row or column area or reorder buttons within the same area.

For illustrations of decision pivots at design time, see the figures in Decision Support Components at Design Time, One-dimensional Crosstab, and Three-dimensional Crosstab.

For information on special properties of TDecisionPivot, see Decision Pivot Properties.

Decision Pivot Properties

The following are some special properties that control the appearance and behavior of decision pivots:

- The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component.
- The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only rows in one location, and a second which contains only columns in another location.
- Typically, *TDecisionPivot* is added above *TDecisionGrid*. In its default orientation, horizontal, buttons on the left side of *TDecisionPivot* apply to fields on the left side of *TDecisionGrid* (rows); buttons on the right side apply to fields at the top of *TDecisionGrid* (columns).
- You can determine where *TDecisionPivot*'s buttons appear by setting its GroupLayout property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default, described in the previous paragraph).

Creating and Using Decision Grids

Decision grid components, TDecisionGrid, present cross-tabulated data in table form. These tables are also called crosstabs, described in About crosstabs. The figure Decision support components at design time shows a decision grid on a form at design time.

The following topics are discussed in this section:

- Creating Decision Grids
- Using Decision Grids
- Decision Grid Properties

Creating Decision Grids

To create a form with one or more tables of cross-tabulated data

1. Follow steps 1–3 listed under Guidelines for using decision support components.
2. Add one or more decision grid components (TDecisionGrid) and bind them to the decision source, TDecisionSource, with the **Object Inspector** by setting their DecisionSource property to the appropriate decision source component.
3. Continue with steps 5–7 listed under Guidelines for using decision support components.

For a description of what appears in the decision grid and how to use it, see Using decision grids..

To add a graph to the form, follow the instructions in Creating decision graphs.

Using Decision Grids

The decision grid component, TDecisionGrid, displays data from decision cubes TDecisionCube bound to decision sources TDecisionSource.

By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can

- Open and Close Dimensions
- Reorganize, or Pivot, Rows and Columns
- Drill Down for Detail
- Limit Dimension Selection to a Single Dimension for Each Axis

For more information about special properties and events of the decision grid, see Decision grid properties.

Opening and Closing Decision Grid Fields

If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (–) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see Decision Grid Properties for details.

Reorganizing Rows and Columns in Decision Grids

You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see Decision Grid Properties for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See Using decision Pivots for instructions.

Drilling Down for Detail in Decision Grids

You can drill down to see more detail in a dimension.

For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want.

To drill down into a dimension,

- Right-click a category label and choose Drill In To This Value, or
- Right-click a pivot button and choose Drilled In.

To make the complete dimension active again,

- Right-click the corresponding pivot button or,
- Right-click the decision grid in the upper-left corner and select the dimension.

Limiting Dimension Selection in Decision Grids

You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see Using Decision Sources.

Decision Grid Properties

The decision grid component, TDecisionGrid, displays data from the TDecisionSource component bound to TDecisionSource. By default, data appears in a grid with category fields on the left side and top of the grid.

The following are some special properties that control the appearance and behavior of decision grids:

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the **Object Inspector**, then select a dimension. Its properties then appear in the **Object Inspector**: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you're through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the **Object Inspector**.
- The *Options* property of TDecisionGrid lets you control display of grid lines (*cgGridLines* = *True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner* = *True*), and enabling of drag-and-drop pivoting (*cgPivotable* = *True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to

the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.

- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is was used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

Creating and Using Decision Graphs

Decision graph components, TDecisionGraph, present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see One-dimensional crosstabs. For illustrations of decision graphs at design time, see the figures Decision support components at design time and Decision graphs bound to different decision sources.

The following topics are discussed in this section:

- Creating Decision Graphs
- Using Decision Graphs
- The Decision Graph Display
- Customizing Decision Graphs

Creating Decision Graphs

To create a form with one or more decision graphs

1. Follow steps 1–3 listed under Guidelines for using decision support components.
2. Add one or more decision graph components TDecisionGraph and bind them to the decision source, TDecisionSource, with the **Object Inspector** by setting their *DecisionSource* property to the appropriate decision source component.
3. Continue with steps 5–7 listed under Guidelines for using decision support components.
4. Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see Customizing decision graphs.

For a description of what appears in the decision graph and how to use it, see Using decision graphs.

To add a decision grid—or crosstab table—to the form, follow the instructions in Creating and using decision grids.

Using Decision Graphs

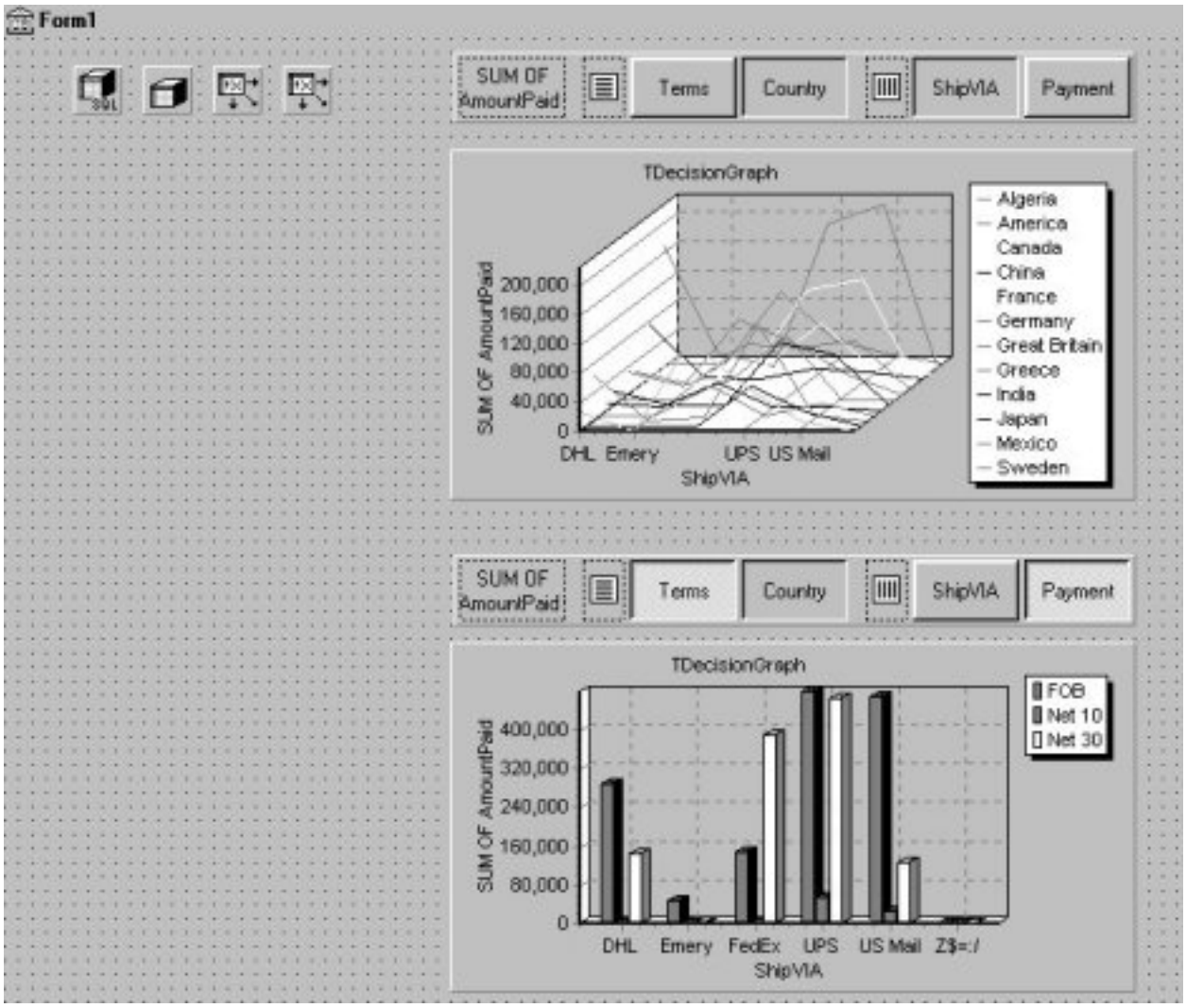
The decision graph component, TDecisionGraph, displays fields from the decision source TDecisionSource as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot.

Graphed data comes from a specially formatted dataset such as TDecisionQuery. For an overview of how the decision support components handle and arrange this data, see Using Decision Support Components.

By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis.

You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data in tabular form. Decision grids and decision graphs that are bound to the same decision source present the same data dimensions. To show different summary data for the same dimensions, you can bind more than one decision graph to the same decision source. To show different dimensions, bind decision graphs to different decision sources.

For example, in the following figure the first decision pivot and graph are bound to the first decision source and the second decision pivot and graph are bound to the second. So, each graph can show different dimensions.



For more information about what appears in a decision graph, see the next section, The Decision Graph Display.

To create a decision graph, see the previous section, Creating Decision Graphs.

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see Customizing Decision Graphs.

The Decision Graph Display

By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series.

If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed.

If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional graph with all buttons on one side of the separator space, you can use the Row or Column icon as a drop target for adding buttons to the other side of the separator and making the graph multidimensional.

If you only want one column and one row to be active at a time, you can set the *ControlType* property for *TDecisionSource* to *xtRadio*. Then, there can be only one active field at a time for each decision cube axis, and the decision pivot's functionality will correspond to the graph's behavior. *xtRadioEx* works the same as *xtRadio*, but does not allow the state where all row or all columns dimensions are closed.

When you have both a decision grid and graph connected to the same *TDecisionSource*, you'll probably want to set *ControlType* back to *xtCheck* to correspond to the more flexible behavior of *TDecisionGrid*.

Customizing Decision Graphs

The decision graph component, TDecisionGraph, displays fields from the decision source (TDecisionSource) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot. You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.

To customize a graph

1. Right-click it and choose Edit Chart. The Chart Editing dialog box appears.
2. Use the Chart page of the Chart Editing dialog box to view a list of visible series, select the series definition to use when two or more are available for the same series, change graph types for a template or series, and set overall graph properties.

   The Series list on the Chart page shows all decision cube dimensions (preceded by Template:) and currently visible categories. Each category, or series, is a separate object. You can:

   - Add or delete series derived from existing decision-graph series. Derived series can provide annotations for existing series or represent values calculated from other series.
   - Change the default graph type, and change the title of templates and series.

3. Use the Series page to establish dimension templates, then customize properties for each individual graph series.

By default, all series are graphed as bar graphs and up to 16 default colors are assigned. You can edit the template type and properties to create a new default. Then, as you pivot the decision source to different states, the template is used to dynamically create the series for each new state. For template details, see Setting decision graph template defaults..

To customize individual series, follow the instructions in Customizing decision graph series.

Setting Decision Graph Template Defaults

Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template.

A separate template is provided for cases where users pivot to a state where only one dimension is active. A one-dimensional state is often represented with a pie chart, so a separate template is provided for this case.

You can

- Change the default graph type.
- Change other graph template properties.
- View and set overall graph properties.

Changing the Default Decision Graph Type

To change the default graph type

1. Select a template in the Series list on the Chart page of the Chart Editing dialog box.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.

Changing Other Decision Graph Template Properties

To change color or other properties of a template

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a template in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.

Viewing Overall Decision Graph Properties

To view and set decision graph properties other than type and series

1. Select the Chart page at the top of the Chart Editing dialog box.
2. Choose the appropriate property tab and select settings.

Customizing Decision Graph Series

The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series.

The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category you want to customize. When the series is displayed on the graph, you can use the Chart editor to

- Change the Graph Type.
- Change Other Series Properties.

- Save Specific Graph Series that You Have Customized.

To define series templates and set overall graph defaults, see Setting Decision Graph Template Defaults.

Changing the Series Graph Type

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See Changing other decision graph series properties. for instructions.

To change the graph type for a single series

1. Select a series in the Series list on the Chart page of the Chart editor.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.
4. Check the Save Series check box.

Changing Other Decision Graph Series Properties

To change color or other properties of a decision graph series

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a series in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.
4. Check the Save Series check box.

Saving Decision Graph Series Settings

By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the **Chart Editing** dialog box.

Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.

Decision Support Components at Runtime

At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations are summarized below.

- Decision Pivots at Runtime
- Decision Grids at Runtime
- Decision Graphs at Runtime

Decision Pivots: Runtime Behavior

Users can:

Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs.

Right-click a dimension button and choose to:

- Move it from the row area to the column area or the reverse.
- Drill In to display detail data.

Left-click a dimension button following the Drill In command and choose:

- Open Dimension to move back to the top level of that dimension.
- All Values to toggle between displaying just summaries and summaries plus all other values in decision grids.
- From a list of available categories for that dimension, a category to drill into for detail values.

Left-click a dimension button to open or close that dimension.

Drag and drop dimension buttons from the row area to the column area and the reverse; they can drop them next to existing buttons in that area or onto the row or column icon.

Decision Grids at Runtime

Users can:

Right-click within the decision grid and choose to:

- Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid.
- Display the Decision Cube editor, described in Using the Decision Cube editor.
- Toggle dimensions and summaries open and closed.

Click + and – within the row and column headings to open and close dimensions.

Drag and drop dimensions from rows to columns and the reverse.

Decision Graphs at Runtime

Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.

Decision Support Components and Memory Control

When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding dimensions to a decision cube can have a dramatic effect on memory use, and can quickly lead to performance problems. This effect is especially pronounced when adding dimensions that have many values.

Memory consumption can be limited by the following techniques:

- Setting maximum Dimensions, Summaries, and Cells
- Setting Dimension State

- Using Paged Dimensions

The decision support components have a number of settings to help you control how and when memory is used. For more information on the properties and techniques mentioned here, see TDecisionCube.

Setting Maximum Dimensions, Summaries, and Cells

The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time.

Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few. For greater control of the absolute memory demands of the decision cube, you can also limit the number of cells in the cube. Set the maximum number of cells on the Cube Capacity page of the Decision Cube editor.

Setting Dimension State

The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active* setting decreases the ability of the cube to manage the available memory.

When *ActiveFlag* is set to *AsNeeded*, a dimension or summary is loaded only if it can be loaded without exceeding the *MaxDimensions*, *MaxSummaries*, or *MaxCells* limit. The decision cube will swap dimensions and summaries that are marked *AsNeeded* in and out of memory to keep within the limits imposed by *MaxCells*, *MaxDimensions*, and *MaxSummaries*. Thus, a dimension or summary may not be loaded in memory if it is not currently being used. Setting dimensions that are not used frequently to *AsNeeded* results in better loading and pivoting performance, although there will be a time delay to access dimensions which are not currently loaded.

Using Paged Dimensions

When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be "paged," or "permanently drilled down." You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened.

It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a time. Information is usually easier to read when displayed this way, and memory consumption is much easier to manage.

Connecting to databases

Connecting to Databases: Overview

Most dataset components can connect directly to a database server. Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and the associated connection components:

Database connection components

Data Access Mechanism	Connection Component
Borland Database Engine (BDE)	TDatabase
ActiveX Data Objects (ADO)	TADOConnection
dbExpress	TSQLConnection
InterBase Express	TIBDatabase

Note: For a discussion of some pros and cons of each of these mechanisms, see Using Databases.

The connection component provides all the information necessary to establish a database connection. This information is different for each type of connection component:

- For information about describing a BDE-based connection, see Identifying the Database.
- For information about describing an ADO-based connection, see Connecting to a Data Store Using TADOConnection.
- For information about describing a dbExpress connection, see Setting up TSQLConnection.
- For information about describing an InterBase Express connection, see TIBDatabase.

Although each type of dataset uses a different connection component, they are all descendants of TCustomConnection. They all perform many of the same tasks and surface many of the same properties, methods, and events.

The following topics discuss many of these common tasks:

- Using Implicit Connections
- Controlling Connections

- Controlling Server Login
- Managing Transactions
- Working with Associated Datasets
- Sending Commands to the Server
- Obtaining Metadata

Using Implicit Connections

No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the DatabaseName property. For ADO-based datasets, you use the ConnectionString property.

When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development, and the default connection you specify can cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own connection components to tune each database connection to your application's needs. Explicit connection components give you greater control. For example, you need to access the connection component to perform the following tasks:

- Customize database server login support. (Implicit connections display a default login dialog to prompt the user for a user name and password.)
- Control transactions and specify transaction isolation levels.
- Execute SQL commands on the server without using a dataset.
- Perform actions on all open datasets that are connected to the same database.

In addition, if you have multiple datasets that all use the same server, it can be easier to use an connection component, so that you only have to specify the server to use in one place. That way, if you later change the server, you do not need to update several dataset components: only the connection component.

Controlling Connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on.

Once you have identified the desired server and any connection parameters, you can use the connection component to explicitly open or close a connection. The connection component generates events when it opens or closes a connection that you can use to customize the response of your application to changes in the database connection.

The following topics provide details about opening and closing database connections:

- Connecting to a Database Server
- Disconnecting From a Database Server

Connecting to a Database Server

There are two ways to connect to a database server using a connection component:

- Call the Open method.
- Set the Connected property to *True*.

Calling the *Open* method sets *Connected* to *True*.

Note: When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's *Open* method.

When you set *Connected* to *True*, the connection component first generates a BeforeConnect event, where you can perform any initialization. For example, you can use this event to alter connection parameters.

After the *BeforeConnect* event, the connection component may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, the connection component generates an AfterConnect event, where you can perform any tasks that require an open connection.

Note: Some connection components generate additional events as well when establishing a connection.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, the connection component drops the connection. Some connection components surface a *KeepConnection* property that allows the connection to remain open even if all the datasets that use it are closed. If *KeepConnection* is *True*, the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to *True* reduces network traffic and speeds up the application. If *KeepConnection* is *False*, the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

Disconnecting from a Database Server

There are two ways to disconnect a server using a connection component:

- Set the Connected property to *False*.
- Call the Close method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, the connection component generates a BeforeDisconnect event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server.

Finally, the connection component generates an AfterDisconnect event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

Note: Calling *Close* or setting *Connected* to *False* disconnects from a database server even if the connection component has a *KeepConnection* property that is *True*.

Controlling Server Login

Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

The first way is to let the default login dialog and processes handle the login. This is the default approach. Set the LoginPrompt property of the connection component to *True* (the default) and add DBLogDlg to the uses clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.

The second way is to supply the login information before the login attempt. Each type of connection component uses a different mechanism for specifying the user name and password:

- For BDE, dbExpress, and InterBase express datasets, the user name and password connection parameters can be accessed through the *Params* property. (For BDE datasets, the parameter values can also be associated with a BDE alias, while for dbExpress datasets, they can also be associated with a connection name).
- For ADO datasets, the user name and password can be included in the *C*onnectionString property (or provided as parameters to the *Open* method).

If you specify the user name and password before the server requests them, be sure to set the *LoginPrompt* to *False*, so that the default login dialog does not appear. For example, the following code sets the user name and password on a SQL connection component in the *BeforeConnect* event handler, decrypting an encrypted password that is associated with the current connection name:

```
[Delphi]
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
```

```
[C++]
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
        Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

Note that setting the user name and password at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. This still leaves them easy to find, compromising server security:

The third way is to provide your own custom handling for the login event. The connection component generates an event when it needs the user name and password.

- For *TDatabase*, *TSQLConnection*, and *TIBDatabase*, this is an *OnLogin* event. The event handler has two parameters, the connection component, and a local copy of the user name and password parameters in a string list. (*TSQLConnection* includes the database parameter as well). You must set the LoginPrompt property to *True* for this event to occur. Having a LoginPrompt value of False and assigning a handler for the OnLogin event

creates a situation where it is impossible to log in to the database because the default dialog does not appear and the OnLogin event handler never executes.

- For *TADOConnection*, the event is an *On*WillConnect event. The event handler has five parameters, the connection component and four parameters that return values to influence the connection (including two for user name and password). This event always occurs, regardless of the value of *LoginPrompt*.

Write an event handler for the event in which you set the login parameters. Here is an example where the values for the USER NAME and PASSWORD parameters are provided from a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*)

```
[Delphi]
procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;
```

```
[C++]
void __fastcall TForm1::Database1Login(TDatabase *Database, TStrings *LoginParams)
{
  LoginParams->Values["USER NAME"] = UserName;
  LoginParams->Values["PASSWORD"] = PasswordSearch(UserName);
}
```

As with the other methods of providing login parameters, when writing an *OnLogin* or *OnWillConnect* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

Managing Transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Most databases provide their own transaction management model, although some have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, connection components provide a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (The BDE also provides limited transaction support for local tables with no server transaction support. When not using the BDE, trying to start transactions on a database that does not support them causes connection components to raise an exception.)

Warning: When a dataset provider component applies updates, it implicitly generates transactions for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

Starting a transaction

When you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or (in the case of overlapping transactions) until another transaction is started. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level.

For *TADOConnection*, start a transaction by calling the *BeginTrans* method:

```
[Delphi]
Level := ADOConnection1.BeginTrans;
```

```
[C++]
Level = ADOConnection1->BeginTrans();
```

BeginTrans returns the level of nesting for the transaction that started. A nested transaction is one that is nested within another, parent, transaction. After the server starts the transaction, the ADO connection receives an *OnBeginTransComplete* event.

For *TDatabase,* use the *StartTransaction* method instead. *TDataBase* does not support nested or overlapped transactions: If you call a *TDatabase* component's *StartTransaction* method while another transaction is underway, it raises an exception. To avoid calling *StartTransaction*, you can check the *InTransaction* property:

```
[Delphi]
if not Database1.InTransaction then
  Database1.StartTransaction;
```

```
[C++]
if (!Database1->InTransaction)
  Database1->StartTransaction();
```

TSQLConnection also uses the *StartTransaction* method, but it uses a version that gives you a lot more control. Specifically, *StartTransaction* takes a transaction descriptor, which lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested (as they can be when using ADO) or they can be overlapped.

```
[Delphi]
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

```
[C++]
TTransactionDesc TD;
TD.TransactionID = 1;
TD.IsolationLevel = xilREADCOMMITTED;
SQLConnection1->StartTransaction(TD);
```

By default, with overlapped transactions, the first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. If you are using *TSQLConnection* with an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its TransactionLevel property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

Note: Unlike *TADOConnection*, *TSQLConnection* and *TDatabase* do not receive any events when the transactions starts.

InterBase express offers you even more control than *TSQLConnection* by using a separate transaction component rather than starting transactions using the connection component. You can, however, use *TIBDatabase* to start a default transaction:

```
[Delphi]
if not IBDatabase1.DefaultTransaction.InTransaction then
  IBDatabase1.DefaultTransaction.StartTransaction;
```

```
[C++]
if (!IBDatabase1->DefaultTransaction->InTransaction)
  IBDatabase1->DefaultTransaction->StartTransaction();
```

You can have overlapped transactions by using two separate transaction components. Each transaction component has a set of parameters that let you configure the transaction. These let you specify the transaction isolation level, as well as other properties of the transaction.

Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by committing the transaction. For *TDatabase*, you commit a transaction using the *Commit* method:

```
[Delphi]
MyOracleConnection.Commit;
```

```
[C++]
MyOracleConnection->Commit();
```

For *TSQLConnection*, you also use the *Commit* method, but you must specify which transaction you are committing by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
[Delphi]
MyOracleConnection.Commit(TD);
```

```
[C++]
MyOracleConnection->Commit(TD);
```

For *TIBDatabase*, you commit a transaction object using its *Commit* method:

```
[Delphi]
IBDatabase1.DefaultTransaction.Commit;
```

```
[C++]
IBDatabase1->DefaultTransaction->Commit();
```

For *TADOConnection*, you commit a transaction using the *CommitTrans* method:

```
[Delphi]
ADOConnection1.CommitTrans;
```

```
[C++]
ADOConnection1->CommitTrans();
```

Note: It is possible for a nested transaction to be committed, only to have the changes rolled back later if the parent transaction is rolled back.

After the transaction is successfully committed, an ADO connection component receives an *OnCommitTransComplete* event. Other connection components do not receive any similar events.

A call to commit the current transaction is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. Discarding these changes is called rolling back the transaction.

For *TDatabase*, you roll back a transaction by calling the *Rollback* method:

```
[Delphi]
MyOracleConnection.Rollback;
```

```
[C++]
MyOracleConnection->Rollback();
```

For *TSQLConnection*, you also use the *Rollback* method, but you must specify which transaction you are rolling back by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
[Delphi]
MyOracleConnection.Rollback(TD);
```

```
[C++]
MyOracleConnection->Rollback(TD);
```

For *TIBDatabase*, you roll back a transaction object by calling its *Rollback* method:

```
[Delphi]
IBDatabase1.DefaultTransaction.Rollback;
```

```
[C++]
IBDatabase1->DefaultTransaction->Rollback();
```

For *TADOConnection*, you roll back a transaction by calling the *RollbackTrans* method:

```
[Delphi]
ADOConnection1.RollbackTrans;
```

```
[C++]
ADOConnection1->RollbackTrans();
```

After the transaction is successfully rolled back, an ADO connection component receives an *OnRollbackTransComplete* event. Other connection components do not receive any similar events.

A call to roll back the current transaction usually occurs in

- Exception handling code when you can't recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Specifying the Transaction Isolation Level

Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels:

- *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle, Sybase, MS-SQL, and InterBase).
- *ReadCommitted*: When the isolation level is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading. This level is available for all transactions except local transactions managed by the BDE.
- *RepeatableRead*: When the isolation level is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions. This level is not available on some servers, such as Sybase and MS-SQL and is unavailable on local transactions managed by the BDE.

In addition, *TSQLConnection* lets you specify database-specific custom isolation levels. Custom isolation levels are defined by the *dbExpress* driver. See your driver documentation for details.

Note: For a detailed description of how each isolation level is implemented, see your server documentation.

TDatabase and *TADOConnection* let you specify the transaction isolation level by setting the *TransIsolation* property. When you set *TransIsolation* to a value that is not supported by the database server, you get the next highest level of isolation (if available). If there is no higher level available, the connection component raises an exception when you try to start a transaction.

When using *TSQLConnection*, transaction isolation level is controlled by the *IsolationLevel* field of the transaction descriptor.

When using InterBase express, transaction isolation level is controlled by a transaction parameter.

Sending Commands to the Server

All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset.

The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL statements also do not return a result set. The DML statements that perform an action on data but do not return a result set are: INSERT, DELETE, and UPDATE.

The syntax for the *Execute* method varies with the connection type:

- For *TDatabase*, *Execute* takes four parameters: a string that specifies a single SQL statement that you want to execute, a TParams object that supplies any parameter values for that statement, a boolean that indicates whether the statement should be cached because you will call it again, and a pointer to a BDE cursor that can be returned (It is recommended that you pass nil).
- For *TADOConnection*, there are two versions of *Execute.* The first takes a WideString that specifies the SQL statement and a second parameter that specifies a set of options that control whether the statement is executed asynchronously and whether it returns any records. This first syntax returns an interface for the returned records. The second syntax takes a WideString that specifies the SQL statement, a second parameter that returns the number of records affected when the statement executes, and a third that specifies options such as whether the statement executes asynchronously. Note that neither syntax provides for passing parameters.
- For *TSQLConnection*, *Execute* takes three parameters: a string that specifies a single SQL statement that you want to execute, a TParams object that supplies any parameter values for that statement, and a pointer that can receive a *TCustomSQLDataSet* that is created to return records.

Note: *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters on a *TSQLConnection* component:

```
[Delphi]
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( " +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

```
[C++]
void __fastcall TDataForm::CreateTableButtonClick(TObject *Sender)
{
  SQLConnection1->Connected = true;
  AnsiString SQLstmt = "CREATE TABLE NewCusts " +
    "( " +
    " CustNo INTEGER, " +
    " Company CHAR(40), " +
    " State CHAR(2), " +
    " PRIMARY KEY (CustNo) " +
    ")";
  SQLConnection1->Execute(SQLstmt, NULL, NULL);
}
```

To use parameters, you must create a TParams object. For each parameter value, use the *TParams*.*CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which uses *TDatabase* to execute an INSERT statement. The INSERT statement has a single parameter named: *StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

```
[Delphi]
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    Database1.Execute(SQLstmt, stmtParams, False, nil);
  finally
    stmtParams.Free;
  end;
end;
```

```
[C++]
void __fastcall TForm1::INSERT_WithParamsButtonClick(TObject *Sender)
{
  AnsiString SQLstmt;
  TParams *stmtParams = new TParams;
  try
  {
    Database1->Connected = true;
    stmtParams->CreateParam(ftString, "StateParam", ptInput);
    stmtParams->ParamByName("StateParam")->AsString = "CA";
    SQLstmt = "INSERT INTO "Custom.db" ";
    SQLstmt += "(CustNo, Company, State) ";
    SQLstmt += "VALUES (7777, "Robin Dabank Consulting", :StateParam)";
    Database1->Execute(SQLstmt, stmtParams, false, NULL);
  }
  __finally
```

```
  {
    delete stmtParams;
  }
}
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to provide its value, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

Working with Associated Datasets

All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.

Closing all datasets without disconnecting from the server

The connection component automatically closes all datasets when you close its connection. There may be times, however, when you want to close all datasets without disconnecting from the database server.

To close all open datasets without disconnecting from a server, you can use the *CloseDataSets* method.

For *TADOConnection* and *TIBDatabase*, calling *CloseDataSets* always leaves the connection open. For *TDatabase* and *TSQLConnection*, you must also set the *KeepConnection* property to *True*.

Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a connection component, use the DataSets and DataSetCount properties. *DataSets* is an indexed array of all datasets that are linked to the connection component. For all connection components except *TADOConnection*, this list includes only the active datasets. *TADOConnection* lists the inactive datasets as well. *DataSetCount* is the number of datasets in this array.

Note: When you use a specialized client dataset to cache updates (as opposed to the generic client dataset, *TClientDataSet*), the *DataSets* property lists the internal dataset owned by the client dataset, not the client dataset itself.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
[Delphi]
var
  I: Integer;
begin
  with MyDBConnection do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

```
[C++]
for (int i = 0; i < MyDBConnection->DataSetCount; i++)
  MyDBConnection->DataSets[i]->DisableControls();
```

Note: *TADOConnection* supports command objects as well as datasets. You can iterate through these much like you iterate through the datasets, by using the *Commands* and *CommandCount* properties.

Obtaining Metadata

All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime.

You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure at runtime.

Listing available tables

The *GetTableNames* method copies a list of table names to an already-existing string list object. This can be used, for example, to fill a list box with table names that the user can then use to choose a table to open. The following line fills a listbox with the names of all tables on the database:

```
[Delphi]
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

```
[C++]
MyDBConnection->GetTableNames(ListBox1->Items, false);
```

GetTableNames has two parameters: the string list to fill with table names, and a boolean that indicates whether the list should include system tables, or ordinary tables. Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.

Note: For most database connection components, *GetTableNames* returns a list of all available non-system tables when the second parameter is *False*. For *TSQLConnection*, however, you have more control over what type is added to the list when you are not fetching only the names of system tables. When using *TSQLConnection*, the types of names added to the list are controlled by the *TableScope* property. *TableScope* indicates whether the list should contain any or all of the following: ordinary tables, system tables, synonyms, and views.

Listing the fields in a table

The *GetFieldNames* method fills an existing string list with the names of all fields (columns) in a specified table. *GetFieldNames* takes two parameters, the name of the table for which you want to list the fields, and an existing string list to be filled with field names:

```
[Delphi]
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

```
[C++]
MyDBConnection->GetTableNames("Employee", ListBox1->Items);
```

Listing available stored procedures

To get a listing of all of the stored procedures contained in the database, use the *GetProcedureNames* method. This method takes a single parameter: an already-existing string list to fill:

```
[Delphi]
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

```
[C++]
MyDBConnection->GetProcedureNames(ListBox1->Items);
```

Note: *GetProcedureNames* is only available for *TADOConnection* and *TSQLConnection*.

Listing available indexes

To get a listing of all indexes defined for a specific table, use the *GetIndexNames* method. This method takes two parameters: the table whose indexes you want, and an already-existing string list to fill:

```
[Delphi]
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

```
[C++]
MyDBConnection1->GetIndexNames("Employee", ListBox1->Items);
```

Note: *GetIndexNames* is only available for *TSQLConnection*, although most table-type datasets have an equivalent method.

Listing stored procedure parameters

To get a list of all parameters defined for a specific stored procedure, use the *GetProcedureParams* method. *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on.

GetProcedureParams takes two parameters: the name of the stored procedure, and an already-existing *TList* object to fill:

```
[Delphi]
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

```
[C++]
MyDBConnection1->GetIndexNames("GetInterestRate", List1);
```

To convert the parameter descriptions that are added to the list into the more familiar *TParams* object, call the global *LoadParamListItems* procedure. Because *GetProcedureParams* dynamically allocates the individual records, your

application must free them when it is finished with the information. The global *FreeProcParams* routine can do this for you.

Note: *GetProcedureParams* is only available for *TSQLConnection*.

Understanding datasets

Understanding Datasets: Overview

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from TDataSet, and they inherit data fields, properties, events, and methods from this class.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An abstract *method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its abstract methods.

TDataSet defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of TField components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *TField* components, see "Working with field components."

The following topics describe how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them. In particular, unidirectional datasets implement only a limited subset.

- Using TDataSet Descendants
- Determining Dataset States
- Opening and Closing Datasets
- Navigating Datasets
- Searching Datasets
- Displaying and Editing a Subset of Data Using Filters
- Modifying Data
- Calculating Fields

- Types of Datasets

Using TDataSet Descendants

TDataSet has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include

- *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of BDE-enabled datasets are described in Using the Borland Database Engine
- *TCustomADODataSet*, which uses ActiveX Data Objects (ADO) to communicate with an OLEDB data store. The *TCustomADODataSet* descendants you use are *TADODataSet*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. The unique features of ADO-based datasets are described in Working with ADO components.
- *TCustomSQLDataSet*, which uses dbExpress to communicate with a database server. The *TCustomSQLDataSet* descendants you use are *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*. The unique features of dbExpress datasets are described in Using Unidirectional Datasets.
- *TIBCustomDataSet*, which communicates directly with an InterBase database server. The *TIBCustomDataSet* descendants you use are *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*.
- *TCustomClientDataSet*, which represents the data from another dataset component or the data from a dedicated file on disk. The *TCustomClientDataSet* descendants you use are *TClientDataSet*, which can connect to an external (source) dataset, and the client datasets that are specialized to a particular data access mechanism (*TBDEClientDataSet, TSimpleDataSet*, and *TIBClientDataSet*), which use an internal source dataset. The unique features of client datasets are described in Using client datasets

Some pros and cons of the various data access mechanisms employed by these *TDataSet* descendants are described in Using databases.

In addition to the built-in datasets, you can create your own custom *TDataSet* descendants—for example to supply data from a process other than a database server, such as a spreadsheet. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see Overview of component creation.

Although each *TDataSet* descendant has its own unique properties and methods, some of the properties and methods introduced by descendant classes are the same as those introduced by other descendant classes that use another data access mechanism. For example, there are similarities between the "table" components (*TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*). For information about the commonalities introduced by *TDataSet* descendants, see Types of datasets.

Determining Dataset States

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only State property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Values for the dataset State property

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
dsInsert	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
dsSetKey	SetKey	DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations. (not supported by all datasets)
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	DataSet open. Indicates that the CurValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsNewValue	NewValue	DataSet open. Indicates that the NewValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsOldValue	OldValue	DataSet open. Indicates that the OldValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
dsBlockRead	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.
dsInternalCalc	Internal Calc	DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only)
dsOpening	Opening	DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Typically, an application checks the dataset state to determine when to perform certain tasks. For example, you might check for the *dsEdit* or *dsInsert* state to ascertain whether you need to post updates.

Note: Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see Responding to Changes Mediated by the Data Source.

Opening and Closing Datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways:

Open Method	Sample Code

Set theActiveproperty of the dataset to *True*, either at design time in the **Object Inspector**, or in code at runtime.	`[Delphi]`
	`CustTable.Active := True;`
	
	`[C++]`
	`CustTable->Active = true;`

Call the Open method for the dataset at runtime.	`[Delphi]` `CustQuery.Open;`

```
[C++]
CustQuery->Open();
```

When you open the dataset, the dataset first receives a BeforeOpen event, then it opens a cursor, populating itself with data, and finally, it receives an AfterOpen event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it.

You can close a dataset in two ways:

Close Method	Sample Code

Set the *Active* property of the dataset to *False*, either at design time in the **Object Inspector**, or in code at runtime.	`[Delphi]`
	`CustQuery.Active := False;`
	
	`[C++]`
	`CustQuery->Active = false;`

Call the Close method for the dataset at runtime.	`[Delphi]`
	`CustTable.Close;`
	
	`[C++]`
	`CustTable->Close();`

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a BeforeClose and AfterClose event when you close it. You can use these events, for example, to prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
[Delphi]

procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
```

```
[C++]
void __fastcall TForm1::VerifyBeforeClose(TDataSet *DataSet)
{
if (DataSet->State == dsEdit || DataSet->State == dsInsert)
{
TMsgDlgButtons btns;
btns << mbYes << mbNo;
    if (MessageDlg("Post changes before closing?", mtConfirmation, btns, 0) == mrYes)
DataSet->Post();
else
DataSet->Cancel();
```

```
}
}
```

Note: You may need to close a dataset when you want to change certain of its properties, such as TableName on a TTable component. When you reopen the dataset, the new property value takes effect.

Navigating Datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as TDBEdit, *TDBLabel*, and TDBMemo. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Navigational methods of datasets

Method	Moves the Cursor to
First	The first row in a dataset.
Last	The last row in a dataset. (not available for unidirectional datasets)
Next	The next row in a dataset.
Prior	The previous row in a dataset. (not available for unidirectional datasets)
MoveBy	A specified number of rows forward or back in a dataset.

The data-aware, visual component TDBNavigator encapsulates these methods as buttons that users can click to move among records at runtime. For information about the navigator component, see Navigating and manipulating records.

Whenever you change the current record using one of these methods (or by other methods that navigate based on a search criterion), the dataset receives two events: BeforeScroll (before leaving the current record) and AfterScroll (after arriving at the new record). You can use these events to update your user interface (for example, to update a status bar that indicates information about the current record).

TDataSet also defines two boolean properties that provide useful information when iterating through the records in a dataset.

Navigational properties of datasets

Property	Description
BOF (Beginning-of-file)	*True*: the cursor is at the first row in the dataset.
	False: the cursor is not known to be at the first row in the dataset
EOF (End-of-file)	*True*: the cursor is at the last row in the dataset.
	False: the cursor is not known to be at the first row in the dataset

The following topics discuss these properties and methods in more detail:

- Using the First and Last methods
- Using the Next and Prior methods
- Using the MoveBy method
- Using the Eof and Bof Properties

- Marking and Returning to Records

Using the First and Last Methods

The First method moves the cursor to the first row in a dataset and sets the BOF property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
[Delphi]
CustTable.First;
```

```
[C++]
CustTable->First();
```

The Last method moves the cursor to the last row in a dataset and sets the EOF property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
[Delphi]
CustTable.Last;
```

```
[C++]
CustTable->Last();
```

Note: The *Last* method raises an exception in unidirectional datasets.

Tip: While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the TDBNavigator component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The OnClick events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see Navigating and manipulating records.

Using the Next and Prior Methods

The Next method moves the cursor forward one row in the dataset and sets the BOF property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
[Delphi]
CustTable.Next;
```

```
[C++]
CustTable->Next();
```

The Prior method moves the cursor back one row in the dataset, and setsEOF to *False* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
[Delphi]
CustTable.Prior;
```

```
[C++]
CustTable->Prior();
```

Note: The *Prior* method raises an exception in unidirectional datasets.

Using the MoveBy Method

MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the BOF and EOF properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

Note: *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
[Delphi]
CustTable.MoveBy(-2);
```

```
[C++]
CustTable->MoveBy(-2);
```

Note: If your application uses *MoveBy* in a multi-user database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back if several users are simultaneously accessing the database and changing its data.

Using the Eof and Bof Properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful when you want to iterate through all records in a dataset.

Eof

When EOF is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.

- Calls *SetRange* on an empty range or dataset.

Eof is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call First) *Eof* is *False*. To iterate through the dataset a record at a time, create a loop that steps through each record by calling Next, and terminates when *Eof* is *True*. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
[Delphi]
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    .
    .
    .
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

```
[C++]
CustTable->DisableControls(); // Speed up processing; prevent screen flicker
try
{
  while (!CustTable->Bof) // Cycle until Bof is true
  (
    // Process each record here
    .
    .
    .
    CustTable->Prior();
    // Bof false on success; Bof true when Prior fails on first record
  }
}
__finally
{
  CustTable->EnableControls();
}
```

```
[C++]
CustTable->DisableControls();
try
{
  for (CustTable->First(); !CustTable->Eof; CustTable->Next())
  (
    // Process each record here
    .
    .
    .
  }
```

```
}
__finally
{
  CustTable->EnableControls();
}
```

Tip: This example also shows how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When BOF is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's First method.
- Calls a dataset's Prior method, and the method fails (because the cursor is currently at the first row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Bof is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like EOF, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
[Delphi]
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    .
    .
    .
    CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

Marking and Returning to Records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. TDataSet introduces a bookmarking feature that consists of a Bookmark property and five bookmark methods.

TDataSet implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. *TDataSet* descendants vary in the level of support they provide for bookmarks. None of the

dbExpress datasets add any support for bookmarks. ADO datasets can support bookmarks, depending on the underlying database tables. BDE datasets, InterBase express datasets, and client datasets always support bookmarks.

The Bookmark property

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

The GetBookmark method

To create a bookmark, you must declare a variable of type *TBookmark* in your application, then call GetBookmark to allocate storage for the variable and set its value to a particular location in a dataset. The *TBookmark* type is a Pointer.

The GotoBookmark and BookmarkValid methods

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. Before calling *GotoBookmark*, you can call BookmarkValid to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record.

The CompareBookmarks method

You can also call CompareBookmarks to see if a bookmark you want to move to is different from another (or the current) bookmark. If the two bookmarks refer to the same record (or if both are **nil**), *CompareBookmarks* returns 0.

The FreeBookmark method

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call FreeBookmark before reusing an existing bookmark.

A bookmarking example

The following code illustrates one use of bookmarking:

```
[Delphi]
procedure DoSomething (const Tbl: TTable)
var
  Bookmark: TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
  Tbl.DisableControls; { Turn off display of records in data controls }
  try
    Tbl.First; { Go to first record in table }
    while not Tbl.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      .
      .
      .
      Tbl.Next;
```

```
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
    Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

```
[C++]
void DoSomething (const TTable *Tbl)
{
  TBookmark Bookmark = Tbl->GetBookmark(); // Allocate memory and assign a value
  Tbl->DisableControls(); // Turn off display of records in data controls
  try
  {
    for (Tbl->First(); !Tbl->Eof; Tbl->Next()) // Iterate through each record in table
    {
      // Do your processing here
      .
      .
      .
    }
  }
  __finally
  {
    Tbl->GotoBookmark(Bookmark);
    Tbl->EnableControls(); // Turn on display of records in data controls
    Tbl->FreeBookmark(Bookmark); // Deallocate memory for the bookmark
  }
}
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

Searching Datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset.

The following topics discuss Locate and Lookup in greater detail:

- Using Locate
- Using Lookup

Note: Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see Using Indexes to Search for Records.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":

```
[Delphi]
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;
```

```
[C++]
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
bool LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the Lookup method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
[Delphi]
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

```
[C++]
TLocateOptions Opts;
Opts.Clear();
Opts << loPartialKey;
Variant locvalues[2];
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("P");
CustTable->Locate("Company;Contact", VarArrayOf(locvalues, 1), Opts);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
[Delphi]
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',     'Company;
Contact; Phone');
end;
```

```
[C++]
Variant LookupResults = CustTable->Lookup("Company", "Professional Divers, Ltd",
    "Company;Contact;Phone");
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
[Delphi]
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
'Company; Addr1; Addr2; State; Zip');
end;
```

```
[C++]
Variant LookupResults;
Variant locvalues[2];
Variant v;
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("Kato Paphos");
LookupResults = CustTable->Lookup("Company;City", VarArrayOf(locvalues, 1),
                  "Company;Addr1;Addr2;State;Zip");
// now put the results in a global stringlist (created elsewhere)
pFieldValues->Clear();
for (int i = 0; i < 5; i++) // Lookup call requested 5 fields
{
v = LookupResults.GetElement(i);
if (v.IsNull())
pFieldValues->Add("");
else
pFieldValues->Add(v);
}
```

Like *Locate*, *Lookup* uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

Displaying and Editing a Subset of Data Using Filters

An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset.

With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's Filter property or coded into its OnFilterRecord event handler. Filter conditions are based on the values in any specified number of fields in a dataset, regardless of whether those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

Note: Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

The following topics describe how to work with filters:

- Enabling and Disabling Filtering
- Navigating Records in a Filtered Dataset

Enabling and Disabling Filtering

To enable filters on a dataset

1. Create a filter.
2. Set filter options for string-based filter tests, if necessary.
3. Set the Filtered property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

Creating Filters

There are two ways to create a filter for a dataset:

- Set the Filter property. Filter is especially useful for creating and applying filters at runtime.
- Write an OnFilterRecord event handler for simple or complex filter conditions. With OnFilterRecord, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the

dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

Setting the Filter Property

To create a filter using the Filter property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
[Delphi]
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

```
[C++]
Dataset1->Filter = "State = 'CA'";
```

You can also supply a value for *Filter* based on text supplied by the user. For example, the following statement assigns the text in from edit box to *Filter*:

```
[Delphi]
Dataset1.Filter := Edit1.Text;
```

```
[C++]
Dataset1->Filter = Edit1->Text;
```

You can, of course, create a string based on both hard-coded text and user-supplied data:

```
[Delphi]
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

```
[C++]
Dataset1->Filter = AnsiString("State = '") + Edit1->Text + "'";
```

Blank field values do not appear unless they are explicitly included in the filter:

```
[Delphi]
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

```
[C++]
Dataset1->Filter = "State <> 'CA' or State = BLANK";
```

Note: After you specify a value for *Filter*, to apply the filter to the dataset, set the Filtered property to *True*.

Filters can compare field values to literals and to constants using the following comparison and logical operators:

Comparison and logical operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than

>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*
+	Adds numbers, concatenates strings, adds numbers to date/time values (only available for some drivers)
-	Subtracts numbers, subtracts dates, or subtracts a number from a date (only available for some drivers)
*	Multiplies two numbers (only available for some drivers)
/	Divides two numbers (only available for some drivers)
*	wildcard for partial comparisons (*FilterOptions* must include *foPartialCompare*)

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Writing an OnFilterRecord Event Handler

You can write code to filter records using the OnFilterRecord events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to "CA":

```
[Delphi]
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

```
[C++]
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet; bool &Accept)
{
  Accept = DataSet->FieldByName["State"]->AsString == "CA";
}
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter's conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly coded as possible to avoid adversely affecting the performance.

You can code any number of *OnFilterRecord* event handlers and switch among them at runtime. For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
[Delphi]
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

```
[C++]
DataSet1->OnFilterRecord = NewYorkFilter;
()
```

Setting Filter Options

The FilterOptions property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

FilterOptions values

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foNoPartialCompare	Disable partial string matching; that is, don't match strings that end with an asterisk (*).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
[Delphi]
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

```
[C++]
TFilterOptions FilterOptions;
FilterOptions->Clear();
FilterOptions << foCaseInsensitive;
Table1->FilterOptions = FilterOptions;
Table1->Filter = "State = "CA"";
```

Navigating Records in a Filtered Dataset

There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Filtered dataset navigational methods

Method	Purpose
FindFirst	Move to the first record that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
FindLast	Move to the last record that matches the current filter criteria.
FindNext	Moves from the current record in the filtered dataset to the next one.
FindPrior	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
[Delphi]
DataSet1.FindFirst;
```

```
[C++]
DataSet1->FindFirst();
```

Provided that you set the Filter property or create an OnFilterRecord event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

Note: If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return False. You can check the status of the Found property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset and you call FindNext, the method returns *False*, and the current record is unchanged.

Modifying Data

You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.)

Dataset methods for inserting, updating, and deleting data

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

The following topics discuss these methods in greater detail:

- Editing Records
- Adding New Records
- Deleting Records
- Posting Data
- Canceling Changes
- Modifying Entire Records

Editing Records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the Edit method to put a dataset into *dsEdit* mode if the read-only CanModify property for the dataset is *True*.

When a dataset transitions to *dsEdit* mode, it first receives a *BeforeEdit* event. After the transition to edit mode is successfully completed, the dataset receives an *After*Edit event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an *OnEditError* event occurs, where you can inform the user of the problem or try to correct the situation that prevented the dataset from entering edit mode.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you have a navigator component on your form, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling Post. You cancel them by calling Cancel. In code, Edit and Post are often used together. For example,

```
[Delphi]
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

```
[C++]
Table1->Edit();
Table1->FieldValues["CustNo"] = 1234;
Table1->Post();
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record. If you are not caching updates, posting writes the change back to the database. If you are caching updates, the change is written to a temporary buffer, where it stays until the dataset's *ApplyUpdates* method is called.

Adding New Records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the Insert or Append methods to put a dataset into *dsInsert* mode if the read-only CanModify property for the dataset is *True*.

When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:

```
[Delphi]
procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
  DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
```

```
[C++]
void __fastcall TForm1::OrdersTableNewRecord(TDataSet *DataSet)
{
  DataSet->FieldByName("OrderDate")->AsDateTime = Date();
}
```

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

Note: Even if a dataset is in *dsInsert* state, adding records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the Next and Last buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call Post when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the database, or, if you are caching updates, *Post* writes the record to an in-memory cache. To write cached inserts and appends to the database, call the dataset's *ApplyUpdates* method.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls Post (or *ApplyUpdates* when using cached updates), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls Post (or *ApplyUpdates* when using cached updates), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.

- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting Records

Use the Delete method to delete the current record in an active dataset. When the *Delete* method is called,

- The dataset receives a *BeforeDelete* event.
- The dataset attempts to delete the current record.
- The dataset returns to the *dsBrowse* state.
- The dataset receives an *AfterDelete* event.

If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:

```
[Delphi]
procedure TForm1.TableBeforeDelete (Dataset: TDataset)
begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

```
[C++]
void __fastcall TForm1::TableBeforeDelete (TDataSet *Dataset)
{
  if (MessageBox(0, "Delete This Record?", "CONFIRM", MB_YESNO) != IDYES)
Abort();
}
```

If *Delete* fails, it generates an *OnDeleteError* event. If the *OnDeleteError* event handler can't correct the problem, the dataset remains in *dsEdit* state. If *Delete* succeeds, the dataset reverts to the *dsBrowse* state and the record that followed the deleted record becomes the current record.

If you are caching updates, the deleted record is not removed from the underlying database table until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting Data

After you finish editing a record, you must call the Post method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates.

- If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state.
- If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits are net written to the database until you call *ApplyUpdates*.
- If the dataset is in the *dsSetKey* state, *Post* returns the dataset to the *dsBrowse* state.

Regardless of the initial state of the dataset, *Post* generates *BeforePost* and *AfterPost* events, before and after writing the current changes. You can use these events to update the user interface, or prevent the dataset from

posting changes by calling the *Abort* procedure. If the call to *Post* fails, the dataset receives an *OnPostError* event, where you can inform the user of the problem or attempt to correct it.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the First, Next, Prior, and Last methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The Append and Insert methods also implicitly post any pending data.

Warning: The Close method does not call *Post* implicitly. Use the BeforeClose event to post any pending edits explicitly.

Canceling Changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called Post. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the Cancel method for the dataset. A call to Cancel always returns a dataset to *dsBrowse* state.

If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current record is restored to its original values.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

Modifying Entire Records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Methods that work with entire records

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit Post.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. The application must perform an explicit *Post*.

These method take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, AppendRecord adds a record to the end of the dataset and InsertRecord inserts a record after the current cursor position. For indexed datasets, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call Edit to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a Post.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, *SetFields* assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a TTable component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
[Delphi]
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

```
[C++]
CountryTable->InsertRecord(ARRAYOFCONST(("Japan", "Tokyo", "Asia")));
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
[Delphi]
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

```
[C++]
TLocateOptions SearchOptions;
SearchOptions->Clear();
SearchOptions << loCaseInsensitive;
if (CountryTable->Locate("Name", "Japan", SearchOptions))
{
  CountryTable->Edit();
  CountryTable->SetFields(ARRAYOFCONST(((void *)NULL, (void *)NULL, (void *)NULL,
        344567, 164700000)));
  CountryTable->Post();
}
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as place holders for the first three columns to preserve their current contents.

Calculating Fields

Using the Fields editor, you can define calculated fields for your datasets. When a dataset contains calculated fields, you provide the code to calculate those field's values in an OnCalcFields event handler.

The AutoCalcFields property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- The dataset enters edit mode.
- A record is retrieved from the database.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a record are edited (the fourth condition above).

Warning: *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or a linked dataset if it is part of a master-detail relationship), because this leads to recursion. For example, if *OnCalcFields* performs a Post, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, causing another Post, and so on.

When *OnCalcFields* executes, a dataset enters *dsCalcFields* mode. This state prevents modifications or additions to the records except for the calculated fields the handler is designed to modify. The reason for preventing other modifications is because *OnCalcFields* uses the values in other fields to derive calculated field values. Changes to those other fields might otherwise invalidate the values assigned to calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Types of Datasets

Using TDataSet descendants classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets:

Table type datasets: Table type datasets represent a single table from the database server, including all of its rows and columns. Table type datasets include TTable, TADOTable, TSQLTable, and TIBTable.

Table type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use server indexes that are defined for the database table. Indexes allow your application to sort the records in the table, speed searches and lookups, and can form the basis of a master/detail relationship. Some table type datasets also take advantage of the one-to-one relationship between dataset and database table to let you perform table-level operations such as creating and deleting database tables.

Query-type datasets: Query-type datasets represent a single SQL command, or query. Queries can represent the result set from executing a command (typically a SELECT statement), or they can execute a command that does not return any records (for example, an UPDATE statement). Query-type datasets include TQuery, TADOQuery, TSQLQuery, and TIBQuery.

To use a query-type dataset effectively, you must be familiar with SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard. If you are new to SQL, you may want to purchase a third party book that covers SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

Stored procedure-type datasets: Stored procedure-type datasets represent a stored procedure on the database server. Stored procedure-type datasets include TStoredProc, TADOStoredProc, TSQLStoredProc, and TIBStoredProc.

A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. They typically handle frequently repeated database-related tasks, and are especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return

data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

Note: You can usually use a query-type dataset to execute stored procedures because most servers provide extensions to SQL for working with stored procedures. Each server, however, uses its own syntax for this. If you choose to use a query-type dataset instead of a stored procedure-type dataset, see your server documentation for the necessary syntax.

In addition to the datasets that fall neatly into these three categories, *TDataSet* has some descendants that fit into more than one category:

- TADODataSet and TSQLDataSethave a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are most similar to query-type datasets, although *TADODataSet* lets you specify an index like a table type dataset.
- TClientDataSet represents the data from another dataset. As such, it can represent a table, query, or stored procedure. *TClientDataSet* behaves most like a table type dataset, because of its index support. However, it also has some of the features of queries and stored procedures: the management of parameters and the ability to execute without retrieving a result set.
- Some other client datasets (like TBDEClientDataSet) have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are like *TClientDataSet*, including parameter support, indexes, and the ability to execute without retrieving a result set.
- TIBDataSet can represent both queries and stored procedures. In fact, it can represent multiple queries and stored procedures simultaneously, with separate properties for each.

Using Table Type Datasets

To use a table type dataset

1. Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.
2. Identify the database server that contains the table you want to use. Each table type dataset does this differently, but typically you specify a database connection component:
   - For *TTable*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
   - For *TADOTable*, specify a *TADOConnection* component using the Connection property.
   - For *TSQLTable*, specify a *TSQLConnection* component using the SQLConnection property.
   - For *TIBTable*, specify a *TIBConnection* component using the Database property.

   For information about using database connection components, see Connecting to databases
3. Set the *TableName* property to the name of the table in the database. You can select tables from a drop-down list if you have already identified a database connection component.
4. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the dataset. The data source component is used to pass a result set from the dataset to data-aware components for display.

Advantages of using table type datasets

The main advantage of using table type datasets is the availability of indexes. Indexes enable your application to

- Sort the Records in the Dataset.

- Locate Records Quickly.
- Limit the Records That are Visible.
- Establish Master/Detail Relationships.

In addition, the one-to-one relationship between table type datasets and database tables enables many of them to be used for

- Controlling Read/Write Access To Tables
- Creating and Deleting Tables
- Emptying Tables
- Synchronizing Tables

Sorting Records with Indexes

An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort (not available on servers that aren't SQL-based).

Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records. On other tables (such as Paradox and dBASE tables), the index is used by the data access mechanism to present records in the desired order.

The following topics provide details on how to obtain information about available indexes and how to specify which index the dataset uses to sort records:

- Obtaining Information about Indexes
- Specifying an Index with IndexName
- Creating an Index with IndexFieldNames

Obtaining Information About Indexes

Your application can obtain information about server-defined indexes from all table type datasets. To obtain a list of available indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:

```
[Delphi]
CustomersTable.GetIndexNames(ListBox1.Items);
```

```
[C++]
CustomersTable->GetIndexNames(ListBox1->Items);
```

Note: For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. You can still change the index back to a primary index on a Paradox table after using an alternative index, however, by setting the *IndexName* property to a blank string.

To obtain information about the fields of the current index, use the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list the field components for the columns that comprise the index.

The following code illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
[Delphi]
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

```
[C++]
AnsiString ListOfIndexFields[20];
for (int i = 0; i < CustomersTable->IndexFieldCount; i++)
  ListOfIndexFields[i] = CustomersTable->IndexFields[i]->FieldName;
```

Note: *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

Specifying an Index with IndexName

Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.)

To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string.

At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the **Object Inspector**. At runtime set *IndexName* using a *String* literal or variable. You can obtain a list of available indexes by calling the *GetIndexNames* method.

The following code sets the index for *CustomersTable* to *CustDescending*:

```
[Delphi]
CustomersTable.IndexName := 'CustDescending';
```

```
[C++]
CustomersTable->IndexName = "CustDescending";
```

For information on specifying dBASE non-production index files and dBASE III PLUS-style .NDX files, see Specifying a dBASE index file

Creating an Index with IndexFieldNames

If there is no defined index that implements the sort order you want, you can create a pseudo-index using the *IndexFieldNames* property.

Note: *IndexName* and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other. For information about *IndexName*, see Specifying an index with IndexName.

The value of *IndexFieldNames* is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
[Delphi]
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

```
[C++]
PhoneTable->IndexFieldNames = "LastName;FirstName";
```

Note: If you use *IndexFieldNames* on Paradox and dBASE tables, the dataset attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

Using Indexes to Search for Records

You can search against any dataset using the Locate and Lookup methods of *TDataSet*. However, by explicitly using indexes, some table type datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

ADO datasets all support the Seek method, which moves to a record based on a set of field values for fields in the current index. *Seek* lets you specify where to move the cursor relative to the first or last matching record.

TTable and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table summarizes the six related methods provided by *TTable* and client datasets to support index-based searches:

Index-based search methods

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the table into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

The following topics discuss the Goto and Find methods in greater detail:

- Executing a Search with Goto Methods
- Executing a Search with Find Methods
- Specifying the Current Record After a Successful Search
- Searching on Partial Keys
- Repeating or Extending a Search

Executing a Search with Goto Methods

To execute a search using Goto methods

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset. To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Put the dataset in *dsSetKey* state by calling the *SetKey* method.
4. Specify the value(s) to search on in the Fields property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, uses the *GotoKey* method to move to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
[Delphi]
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

```
[C++]
void __fastcall TSearchDemo::SearchExactClick(TObject *Sender)
{
  ClientDataSet1->SetKey();
  ClientDataSet1->Fields->Fields[0]->AsString = Edit1->Text;
  if (!ClientDataSet1->GotoKey())
    ShowMessage("Record not found");
}
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
[Delphi]
Table1.SetKey;
```

```
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

```
[C++]
Table1->SetKey();
Table1->Fields->Fields[0]->AsString = "Sm";
Table1->GotoNearest();
```

If a record exists with "Sm" as the first two characters of the first indexed field's value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

Executing a Search with Find Methods

The *Find* methods do the same thing as the Goto methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search.

To execute a search using Find methods

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset. To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

Note: *FindNearest* can only be used for string fields.

Specifying the Current Record After a Successful Search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

Searching On Partial Keys

If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For table type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Searching On Partial Keys

Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*.

For example, suppose you have already executed a search of the Employee table based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
[Delphi]
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

```
[C++]
Employee->KeyFieldCount = 2;
Employee->EditKey();
Employee->FieldValues["Company"] = Variant(Edit2->Text);
Employee->GotoNearest();
```

Limiting Records with Ranges

You can temporarily view and edit a subset of data for any dataset by using filters. Some table type datasets support an additional way to access a subset of available records, called ranges.

Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges:

- Understanding the differences between ranges and filters
- Specifying ranges
- Modifying a range
- Applying or canceling a range

Understanding the Differences Between Ranges and Filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query-type dataset to select data. For details on specifying a query, see Using query-type datasets.

Specifying Ranges

There are two mutually exclusive ways to specify a range:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.

- Specify both endpoints at once using *SetRange*.

Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the Fields property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a *TSimpleDataSet* component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
[Delphi]
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

```
[C++]
Customers->SetRangeStart();
Customers->FieldValues["CustNo"] = StrToInt(StartVal->Text);
Customers->SetRangeEnd();
if (!EndVal->Text.IsEmpty())
  Customers->FieldValues["CustNo"] = StrToInt(EndVal->Text);
Customers->ApplyRange();
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

Tip: To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range.

Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the Fields property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

Warning: Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the FieldByName method. For example,

```
[Delphi]
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

```
[C++]
Contacts->SetRangeStart();
Contacts->FieldByName("LastName")->Value = Edit1->Text;
Contacts->SetRangeEnd();
Contacts->FieldByName("LastName")->Value = Edit2->Text;
Contacts->ApplyRange();
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range.

Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statement establishes a range based on a two-column index:

```
[Delphi]
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

```
[C++]
TVarRec StartVals[2];
TVarRec EndVals[2];
StartVals[0] = Edit1->Text;
StartVals[1] = Edit2->Text;
EndVals[0] = Edit3->Text;
EndVals[1] = Edit4->Text;
Table1->SetRange(StartVals, 1, EndVals, 1);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
[Delphi]
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

```
[C++]
Contacts->SetRangeStart();
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Zzzzzz";
Contacts->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith." The value specification could also be:

```
[Delphi]
Contacts['LastName'] := 'Sm';
```

```
[C++]
Contacts->FieldValues["LastName"] = "Sm";
```

This statement includes records that have *LastName* greater than or equal to "Sm."

Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a dataset to *True* to exclude records equal to ending range. For example,

```
[Delphi]
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

```
[C++]
Contacts->SetRangeStart();
```

```
Contacts->KeyExclusive = true;
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Tyler";
Contacts->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith" and less than "Tyler".

Modifying a Range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

To edit and apply a range

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the Fields property overwrite the current index values to use when applying the range.

Tip: If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range.

Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the Fields property are treated as ending index values to use when applying the range.

Applying or Canceling a Range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* method. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
[Delphi]
.
.
.
MyTable.CancelRange;
.
.
.
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
.
.
.
```

```
[C++]
.
.
.
MyTable->CancelRange();
.
.
.
// later on, use the same range again. No need to call SetRangeStart, etc.
MyTable->ApplyRange();
.
.
.
```

Creating Master/detail Relationships

Table type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Table type datasets support master/detail relationships in two very distinct ways:

- All table type datasets can act as the detail of another dataset by linking cursors. This process is described in Making the table a detail of another dataset.
- *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses nested detail tables. This process is described in Using nested detail tables.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is any type of dataset. With nested details, the type of dataset that can act as the detail table is limited, but they provide for more options in how to display the data. If the master is a client dataset, nested details provide a more robust mechanism for applying cached updates.

Making the Table a Detail of Another Dataset

A table type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query.

The dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the detail dataset, first specify the index in the detail dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you specify the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the index fields in the detail table. To link datasets on multiple column names, separate field names with semicolons:

```
[Delphi]
Parts.MasterFields := 'OrderNo;ItemNo';
```

```
[C++]
Parts->MasterFields = "OrderNo;ItemNo";
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the **Object Inspector** after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the CustomersTable table, and the detail table is OrdersTable. The example uses the BDE-based TTable component, but you can use the same methods to link any table type datasets.

To create a simple form

1. Place two *TTable* components and two *TDataSource* components in a data module.
2. Set the properties of the following components,

Component	Property
First *TTable*	*DatabaseName*: DBDEMOS *TableName*: CUSTOMER *Name*: CustomersTable
Second *TTable*	*DatabaseName*: DBDEMOS *TableName*: ORDERS *Name*: OrdersTable
First *TDataSource*	*Name*: CustSource *DataSet*: CustomersTable
Second *TDataSource*	*Name*: OrdersSource *DataSet*: OrdersTable

3. Place two *TDBGrid* components on a form.
4. Choose **File** ▸ **Use Unit** to specify that the form should use the data module.

5 Set the DataSource property of the first grid component to "CustSource", and set the *DataSource* property of the second grid to "OrdersSource".

6 Set the MasterSource property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).

7 Double-click the MasterFields property value box in the **Object Inspector** to invoke the Field Link Designer to set the following properties:

- In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
- Select *CustNo* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list,"CustNo -> CustNo" appears.
- Choose OK to commit your selections and exit the Field Link Designer.

8 Set the Active properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.

9 Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

Using Nested Detail Tables

A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship.

Note: For *TClientDataSet*, using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server.

To use nested detail sets, the ObjectView property of the master dataset must be *True*. When your table type dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see Working with dataset fields.

Alternately, you can display and edit detail datasets in data-aware controls by using a separate dataset component for the detail set. At design time, create persistent fields for the fields in your (master) dataset, using the Fields Editor: right click the master dataset and choose Fields Editor. Add a new persistent field to your dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of the detail table. You must also add persistent fields for any other fields used in your master dataset.

The dataset component for the detail table is a dataset descendant of a type allowed by the master table. *TTable* components only allow *TNestedDataSet* components as nested datasets. *TSQLTable* components allow other *TSQLTable* components. *TClientDataset* components allow other client datasets. Choose a dataset of the appropriate type from the **Tool palette** and add it to your form or data module. Set this detail dataset's DataSetField property to the persistent DataSet field in the master dataset. Finally, place a data source component on the form or data module and set its *DataSet* property to the detail dataset. Data-aware controls can use this data source to access the data in the detail set.

Controlling Read/Write Access to Tables

By default when a table type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted

(for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

Note: This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset, and hence always read-only.

When the table opens, you can check the CanModify property to ascertain whether the underlying database (or the dataset provider) allows users to edit the data in the table. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided the table's *ReadOnly* property is *False*.

ReadOnly determines whether a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening the table.

Note: *ReadOnly* is implemented on all table type datasets except *TSQLTable*, which is always read-only.

Creating and Deleting Tables

Some table type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.

Creating tables

TTable and *TIBTable* both let you create the underlying database table without using SQL. Similarly, *TClientDataSet* lets you create a dataset when you are not working with a dataset provider. Using *TTable* and *TClientDataSet*, you can create the table at design time or runtime. *TIBTable* only lets you create tables at runtime.

Before you can create the table, you must be set properties to specify the structure of the table you are creating. In particular, you must specify

- The database that will host the new table. For *TTable*, you specify the database using the DatabaseName property. For *TIBTable*, you must use a *TIBDatabase* component, which is assigned to the Database property. (Client datasets do not use a database.)
- The type of database (*TTable* only). Set the TableType property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.
- The name of the table you want to create. Both *TTable* and *TIBTable* have a *TableName* property for the name of the new table. Client datasets do not use a table name, but you should specify the *FileName* property before you save the new table. If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered. To avoid overwriting an existing table, you can check the *Exists* property at runtime. *Exists* is only available on *TTable* and *TIBTable*.
- Indexes for the new table (optional). At design time, double-click the *IndexDefs* property in the **Object Inspector** to bring up the collection editor. Use the collection editor to add, remove, or change the properties of index definitions. At runtime, clear any existing index definitions, and then use the AddIndexDef method to add each new index definition. For each new index definition, set the properties of the TIndexDef object to specify the desired attributes of the index.
- The fields for the new table. There are two ways to do this:
- You can add field definitions to the FieldDefs property. At design time, double-click the *FieldDefs* property in the **Object Inspector** to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the AddFieldDef

method to add each new field definition. For each new field definition, set the properties of the TFieldDef object to specify the desired attributes of the field.

- You can use persistent field components instead. At design time, double-click on the dataset to bring up the *Fields* editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field. Once the field is created, you can alter its properties in the **Object Inspector** by selecting the field in the Fields editor.

Note: You can't define indexes for the new table if you are using persistent field components instead of field definition objects.

To create the table at design time, right-click the dataset and choose Create Table (*TTable*) or Create Data Set (*TClientDataSet*). This command does not appear on the context menu until you have specified all the necessary information.

To create the table at runtime, call the *CreateTable* method (*TTable* and *TIBTable*) or the *CreateDataSet* method (*TClientDataSet*).

Note: You can set up the definitions at design time and then call the *CreateTable* (or *CreateDataSet*) method at runtime to create the table. However, to do so you must indicate that the definitions specified at runtime should be saved with the dataset component. (by default, field and index definitions are generated dynamically at runtime). Specify that the definitions should be saved with the dataset by setting its *StoreDefs* property to *True*.

Tip: If you are using *TTable*, you can preload the field definitions and index definitions of an existing table at design time. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table.

Note: When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, and dataset fields).

The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
[Delphi]
var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
  begin
    try
      { set properties of the temporary TTable component }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { define fields for the new table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
        Name := 'Second';
        DataType := ftString;
```

```
        Size := 30;
        Required := False;
      end;
      { define indexes for the new table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // check whether the table already exists
      if TableFound then
        if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
             mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // create the table
    finally
      Free; // destroy the temporary TTable when done
    end;
  end;
end;
```

```
[C++]
TTable *NewTable = new TTable(Form1);
NewTable->Active = false;
NewTable->DatabaseName = "BCDEMOS";
NewTable->TableName = Edit1->Text;
NewTable->TableType = ttDefault;
NewTable->FieldDefs->Clear();
TFieldDef *NewField = NewTable->FieldDefs->AddFieldDef(); // define first field
NewField->DataType = ftInteger;
NewField->Name = Edit2->Text;
NewField = NewTable->FieldDefs->AddFieldDef(); // define second field
NewField->DataType = ftString;
NewField->Size = StrToInt(Edit3->Text);
NewField->Name = Edit4->Text;
NewTable->IndexDefs->Clear();
TIndexDef *NewIndex = NewTable->IndexDefs->AddIndexDef(); // add an index
NewIndex->Name = "PrimaryIndex";
NewIndex->Fields = Edit2->Text;
NewIndex->Options << ixPrimary << ixUnique;
// Now check for prior existence of this table
bool CreateIt = (!NewTable->Exists);
if (!CreateIt)
   if (Application->MessageBox((AnsiString("Overwrite table ") + Edit1->Text +
                               AnsiString("?")).c_str(),
                               "Table Exists", MB_YESNO) == IDYES)
    CreateIt = true;
if (CreateIt)
  NewTable->CreateTable(); // create the table
```

Deleting tables

TTable and *TIBTable* let you delete tables from the underlying database table without using SQL. To delete a table at runtime, call the dataset's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
[Delphi]
CustomersTable.DeleteTable;
```

```
[C++]
CustomersTable->DeleteTable();
```

Warning: When you delete a table with *DeleteTable*, the table and all its data are gone forever.

If you are using *TTable*, you can also delete tables at design time: Right-click the table component and select Delete Table from the context menu. The Delete Table menu pick is only present if the table component represents an existing database table (the DatabaseName and TableName properties specify an existing table).

Emptying Tables

Many table type datasets supply a single method that lets you delete all rows of data in the table.

Table Type	Method

TTable and *TIBTable*	You can delete all the records by calling the *EmptyTable* method at runtime:
	`[Delphi]`
	`PhoneTable.EmptyTable;`
	`[C++]`
	`PhoneTable->EmptyTable();`

TADOTable	You can use the *DeleteRecords* method:
	`[Delphi]`
	`PhoneTable.DeleteRecords;`
	`[C++]`
	`PhoneTable->DeleteRecords(arAll);`

TSQLTable	You can use the *DeleteRecords* method. Note, that the *TSQLTable* version of *DeleteRecords* never takes any parameters:
	`[Delphi]`
	`PhoneTable.DeleteRecords;`
	`[C++]`
	`PhoneTable->DeleteRecords();`

EmptyDataSet	For client datasets, you can use the *EmptyDataSet* method:
	`[Delphi]`
	`PhoneTable.EmptyDataSet;`
	`[C++]`
	`PhoneTable->EmptyDataSet();`

Note: For tables on SQL servers, these methods only succeed if you have DELETE privilege for the table.

Warning: When you empty a dataset, the data you delete is gone forever.

Synchronizing Tables

If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ.

If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of a matching dataset. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
[Delphi]
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

```
[C++]
CustomerTableOne->GotoCurrent(CustomerTableTwo);
```

Tip: If your application needs to synchronize datasets in this manner, put the datasets in a data module and add the unit for the data module to the uses clause of each unit that accesses the tables.

To synchronize datasets from separate forms, you must add one form's unit to the uses clause of the other, and you must qualify at least one of the dataset names with its form name. For example:

```
[Delphi]
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

```
[C++]
CustomerTableOne->GotoCurrent(Form2->CustomerTableTwo);
```

Using Query-type Datasets

To use a query-type dataset

1 Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.

2 Identify the database server to query. Each query-type dataset does this differently, but typically you specify a database connection component:

- For *TQuery*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
- For *TADOQuery*, specify a *TADOConnection* component using the Connection property.
- For *TSQLQuery*, specify a *TSQLConnection* component using the SQLConnection property.
- For *TIBQuery*, specify a *TIBConnection* component using the Database property.

For information about using database connection components, see Connecting to databases

3 Specify an SQL statement in the *SQL* property of the dataset, and optionally specify any parameters for the statement.

4 If the query data is to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the query-type dataset. The data source component forwards the results of the query (called a *result set*) to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. To execute queries that only perform an action on a table and return no result set, use the *ExecSQL* method at runtime. If you plan to execute the query more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the query. For information about preparing a query, see Preparing queries.

In addition to the basic steps described above, the following topics describe how to establish master/detsil relationships when using query-type datasets and how to improve performance when you only need a unidirectional cursor:

- Establishing master/detail relationships using parameters
- Using unidirectional result sets

Specifying the Query

For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet,* and client datasets, use a *CommandText* property to accomplish the same thing.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Queries that do not return records include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution. In most cases, the SQL command must be only one complete SQL statement, although that statement can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some

servers also support "batch" syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements when you specify the query.

The SQL statements used by queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at runtime without having to alter the SQL statement. For more information about parameterized queries, see Using parameters in queries.

Specifying a query using the SQL property

When using a true query-type dataset *(TQuery, TADOQuery, TSQLQuery,* or *TIBQuery)*, assign the query to the *SQL* property. The *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to modify and debug the query if you divide the statement into logical units:

```
[Delphi]
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
```

```
[C++]
MyQuery->Close();
MyQuery->SQL->Clear();
MyQuery->SQL->Add("SELECT CustNo, OrderNO, SaleDate");
MyQuery->SQL->Add("FROM Orders");
MyQuery->SQL->Add("ORDER BY SaleDate");
MyQuery->Open();
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the ORDER BY clause already exists on the third line of the statement. It is referenced via the *SQL* property using an index of 2.

```
[Delphi]
MyQuery.SQL[2] := 'ORDER BY OrderNo';
```

```
[C++]
MyQuery->SQL->Strings[2] = "ORDER BY OrderNO";
```

Note: The dataset must be closed when you specify or modify the SQL property.

At design time, use the String List editor to specify the query. Click the ellipsis button by the *SQL* property in the **Object Inspector** to display the String List editor.

Note: With some versions of Delphi, if you are using *TQuery*, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select the query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use SQL Builder, open it and use its online help.

Because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings*. *LoadFromFile* method:

```
[Delphi]
MyQuery.SQL.LoadFromFile('custquery.sql');
```

```
[C++]
MyQuery->SQL->LoadFromFile("custquery.sql");
```

You can also use the *Assign* method of the *SQL* property to copy the contents of a string list object into the *SQL* property. The *Assign* method automatically clears the current contents of the *SQL* property before copying the new statement:

```
[Delphi]
MyQuery.SQL.Assign(Memo1.Lines);
```

```
[C++]
MyQuery->SQL->Assign(Memo1->Lines);
```

Specifying a query using the CommandText property

When using *TADODataSet*, *TSQLDataSet,* or a client dataset, assign the text of the query statement to the *CommandText* property:

```
[Delphi]
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
```

```
[C++]
MyQuery->CommandText = "SELECT CustName, Address FROM Customer";
```

At design time, you can type the query directly into the **Object Inspector**, or, if the dataset already has an active connection to the database, you can click the ellipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

Using Parameters in Queries

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:Name*, *:Capital*, and *:Population* are placeholders for actual values supplied to the statement at runtime by your application. Note that the names of parameters begin with a colon. The colon is required so that the parameter names can be distinguished from literal values. You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. *TQuery*, *TIBQuery*, *TSQLQuery*, and client datasets use the *P*arams property to store these values. *TADOQuery* uses the *Parameters* property instead. *Params* (or *Parame*ters) is a collection of parameter objects (TParam or TParameter),

where each object represents a single parameter. When you specify the text for the query, the dataset generates this set of parameter objects, and (depending on the dataset type) initializes any of their properties that it can deduce from the query.

Note: You can suppress the automatic generation of parameter objects in response to changing the query text by setting the *ParamCheck* property to *False*. This is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement that are not parameters for the query itself. For example, the DDL statement to create a stored procedure may define parameters that are part of the stored procedure. By setting *ParamCheck* to *False*, you prevent these parameters from being mistaken for parameters of the query.

Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

Tip: It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names is especially important if the dataset uses a data source to obtain parameter values from another dataset. This process is described in Establishing master/detail relationships using parameters.

The following topics describe how to specify the datatypes and values of parameters for your query:

- Supplying Parameters at Design Time
- Supplying Parameters at Runtime

Supplying Parameters at Design Time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the **Object Inspector**. If the SQL statement does not contain any parameters, no objects are listed in the collection editor.

Note: The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the **Object Inspector** to modify its properties.

When using the *Params* property (*TParam* objects), you will want to inspect or modify the following,

The DataType property lists the data type for the parameter's value. For some datasets, this value may be correctly initialized. If the dataset did not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.

The *DataType* property lists the logical data type for the parameter. In general, these data types conform to server data types. For specific logical type-to-server data type mappings, see the documentation for the data access mechanism (BDE, dbExpress, InterBase).

The ParamType property lists the type of the selected parameter. For queries, this is always *ptInput*, because queries can only contain input parameters. If the value of *ParamType* is *ptUnknown*, change it to *ptInput*.

The Value property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

When using the *Parameters* property (*TParameter* objects), you will want to inspect or modify the following:

The DataType property lists the data type for the parameter's value. For some data types, you must provide additional information:

- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.

The Direction property lists the type of the selected parameter. For queries, this is always *pdInput*, because queries can only contain input parameters.

The Attributes property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.

The Value property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

Supplying Parameters at Runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*)
- *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* or *Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
[Delphi]
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

```
[C++]
SQLQuery1->ParamByName("Capital")->AsString = Edit1->Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
[Delphi]
SQLQuery1.Params[0].AsString := Edit1.Text;
```

```
[C++]
SQLQuery1->Params->Items[0]->AsString = Edit1->Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
[Delphi]
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

```
[C++]
Query1->Params->ParamValues["Name;Capital;Continent"] =
  VarArrayOf(OPENARRAY(Variant, (Edit1->Text, Edit2->Text, Edit3->Text)));
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

Establishing Master/detail Relationships Using Parameters

To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the DataSource property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, query-type datasets attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

To set up the Customer dataset

1. Add a table type dataset to your application and bind it to the Customer table.
2. Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
3. Add a query-type dataset and set its *SQL* property to

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

   Note that the name of the parameter is the same as the name of the field in the master (Customer) table.

4. Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail datasetis not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomersSource*. *CustomersSource* gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called "ID," the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset's SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset's SELECT statement executes to retrieve all orders based on the current customer id.

Preparing Queries

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though the

dataset prepares queries for you, you can improve performance by explicitly preparing the dataset before you open it the first time.

```
[Delphi]
CustQuery.Prepared := True;
```

```
[C++]
CustQuery->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you add a parameter).

Note: When you change the text of the *SQL* property for a query, the dataset automatically closes and unprepares the query.

Executing Queries That Don't Return a Result Set

When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records).

For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL*:

```
[Delphi]
CustomerQuery.ExecSQL;  { query does not return a result set }
```

```
[C++]
CustomerQuery->ExecSQL(); // Does not return a result set
```

Tip: If you are executing the query multiple times, it is a good idea to set the *Prepared* property to *True*.

Although the query does not return any records, you may want to know the number of records it affected (for example, the number of records deleted by a DELETE query). The *RowsAffected* property gives the number of affected records after a call to *ExecSQL*.

Tip: When you do not know at design time whether the query returns a result set (for example, if the user supplies the query dynamically at runtime), you can code both types of query execution statements in a **try...except** block. Put a call to the *Open* method in the **try** clause. An action query is executed when the query is activated with the *Open* method, but an exception occurs in addition to that. Check the exception, and suppress it if it merely indicates the lack of a result set. (For example, *TQuery* indicates this by an *ENoResultSet* exception.)

Using Unidirectional Result Sets

When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using dbExpress, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward

through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries.

If you do not need to be able to navigate backward through a result set, *TQuery* and *TIBQuery* let you improve query performance by requesting a unidirectional cursor instead. To request a unidirectional cursor, set the *UniDirectional* property to True.

Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
[Delphi]
if not (CustomerQuery.Prepared) then
begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepared := True;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

```
[C++]
if (!CustomerQuery->Prepared)
{
  CustomerQuery->UniDirectional = true;
CustomerQuery->Prepared = true;
}
CustomerQuery->Open(); // Returns a result set with a one-way cursor
```

Note: Do not confuse the *UniDirectional* property with a unidirectional dataset. Unidirectional datasets (*TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*) use dbExpress, which only returns unidirectional cursors. In addition to restricting the ability to navigate backwards, unidirectional datasets do not buffer records, and so have additional limitations (such as the inability to use filters).

Using Stored Procedure-type Datasets

How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

To access a stored procedure on a server

1. Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.
2. Identify the database server that defines the stored procedure. Each stored procedure-type dataset does this differently, but typically you specify a database connection component:
   - For *TStoredProc*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
   - For *TADOStoredProc*, specify a *TADOConnection* component using the Connection property.
   - For *TSQLStoredProc*, specify a *TSQLConnection* component using the SQLConnection property.
   - For *TIBStoredProc*, specify a *TIBConnection* component using the Database property.

   For information about using database connection components, see Connecting to databases
3. Specify the stored procedure to execute. For most stored procedure-type datasets, you do this by setting the *StoredProcName* property. The one exception is *TADOStoredProc*, which has a *ProcedureName* property instead.

4 If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the stored procedure-type dataset. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 Provide input parameter values for the stored procedure, if necessary. If the server does not provide information about all stored procedure parameters, you may need to provide additional input parameter information, such as parameter names and data types. For information about working with stored procedure parameters, see Working with stored procedure parameters.

6 Execute the stored procedure. For stored procedures that return a cursor, use the *Active* property or the *Open* method. To execute stored procedures that do not return any results or that only return output parameters, use the *ExecProc* method at runtime. If you plan to execute the stored procedure more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the stored procedure. For information about preparing a query, see Preparing stored procedures.

7 Process any results. These results can be returned as result and output parameters, or they can be returned as a result set that populates the stored procedure-type dataset. Some stored procedures return multiple cursors. For details on how to access the additional cursors, see Fetching multiple result sets.

Working with Stored Procedure Parameters

There are four types of parameters that can be associated with stored procedures:

- *Input p*arameters, used to pass values to a stored procedure for processing.
- *Output* parameters, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- *A result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by the *Params* property (in *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) or the *Parameters* property (in *TADOStoredProc*). When you assign a value to the *StoredProcName* (or *ProcedureName*) property, the dataset automatically generates objects for each parameter of the stored procedure. For some datasets, if the stored procedure name is not specified until runtime, objects for each parameter must be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* or *TParameter* objects allows a single dataset to be used with any number of available stored procedures.

Note: Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters.

Setting up parameters at design time

You can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the **Object Inspector**.

Warning: You can assign values to input parameters by selecting them in the parameter collection editor and using the **Object Inspector** to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add, you must fully describe the parameter. Even if you do not need to add any parameters, you should check the properties of individual parameter objects to ensure that they are correct.

If the dataset has a *Params* property (*TParam* objects), the following properties must be correctly specified:

- The Name property indicates the name of the parameter as it is defined by the stored procedure.
- The DataType property gives the data type for the parameter's value. When using *TSQLStoredProc*, some data types require additional information:
- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.
- The ParamType property indicates the type of the selected parameter. This can be *ptInput* (for input parameters), *ptOutput* (for output parameters), *ptInputOutput* (for input/output parameters) or *ptResult* (for result parameters).
- The Value property specifies a value for the selected parameter. You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

If the dataset uses a *Parameters* property (*TParameter* objects), the following properties must be correctly specified:

- The Name property indicates the name of the parameter as it is defined by the stored procedure.
- The DataType property gives the data type for the parameter's value. For some data types, you must provide additional information:
- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.
- The Direction property gives the type of the selected parameter. This can be *pdInput* (for input parameters), *pdOutput* (for output parameters), *pdInputOutput* (for input/output parameters) or *pdReturnValue* (for result parameters).
- The Attributes property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The Value property specifies a value for the selected parameter. Do not set values for output and result parameters. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

Using parameters at runtime

With some datasets, if the name of the stored procedure is not specified until runtime, no *TParam* objects are automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method:

```
[Delphi]
var
```

```
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
      ParamByname('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
```

```
[C++]
TParam *P1, *P2;
StoredProc1->StoredProcName = "GET_EMP_PROJ";
StoredProc1->Params->Clear();
P1 = new TParam(StoredProc1->Params, ptInput);
P2 = new TParam(StoredProc1->Params, ptOutput);
try
{
  StoredProc1->Params->Items[0]->Name = "EMP_NO";
  StoredProc1->Params->Items[1]->Name = "PROJ_ID";
  StoredProc1->ParamByName("EMP_NO")->AsSmallInt = 52;
  StoredProc1->ExecProc();
  Edit1->Text = StoredProc1->ParamByName("PROJ_ID")->AsString;
}
__finally
{
  delete P1;
  delete P2;
}
```

Even if you do not need to add the individual parameter objects at runtime, you may want to access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. You can use the dataset's *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

```
[Delphi]
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

```
[C++]
SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger = 103;
```

```
SQLDataSet1->ExecSQL();
int Result = SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger;
```

Preparing Stored Procedures

As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance.

If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the *Prepared* property to *True*.

```
[Delphi]
MyProc.Prepared := True;
```

```
[C++]
MyProc->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the stored procedure are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change the parameters when using Oracle overloaded procedures).

Executing Stored Procedures That Don't Return a Result Set

When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

```
[Delphi]
MyStoredProcedure.ExecProc;  { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

```
[C++]
MyStoredProcedure->ExecProc(); // Does not return a result set
Edit1->Text = MyStoredProcedure->ParamByName("OUTVAR")->AsString;
```

Note: *TADOStoredProc* does not have a *ParamByName* method. To obtain output parameter values when using ADO, access parameter objects using the *Parameters* property.

Tip: If you are executing the procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

Fetching Multiple Result Sets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:

```
[Delphi]
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

```
[C++]
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

In *TSQLStoredProc, NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. In *TADOStoredProc*, *NextRecordset* returns an interface that can be assigned to the *RecordSet* property of an existing ADO dataset. For either class, the method returns the number of records in the returned dataset as an output parameter.

The first time you call *NextRecordSet*, it returns the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional cursors, *NextRecordSet* returns **nil**.

Working with field components

Working with Field Components: Overview

Field components represent individual fields (columns) in datasets. You can use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a TField object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as TDBEdit and TDBGrid access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a TFloatField component has four properties that directly affect the appearance of its data:

TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as DisplayWidth and *Alignment*), and they have properties specific to their data types (such as Precision for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications.

The following topics discuss field components in greater detail:

- Dynamic Field Components
- Persistent Field Components
- Working with Field Component Methods at Runtime

- Displaying, Converting, and Accessing Field Values
- Setting a Default Value for a Field
- Working with Constraints
- Using Object Fields

Dynamic Field Components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact TField descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.
4. Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any field name you specify will exist when the dataset is opened.
5. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

Persistent Field Components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.

- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note: When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see Dynamic field components.

Note: One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see Creating a customized grid.

The following topics describe how to use the Fields editor to create or modify the persistent fields in a dataset, and how to work with persistent fields:

- Creating Persistent Fields
- Arranging Persistent Fields
- Defining New Persistent Fields
- Deleting Persistent Field Components
- Setting Persistent Field Properties and Events

Creating Persistent Fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset

1. Place a dataset in a data module.
2. Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TADODataSet*, you can set the *Connection* property to a properly configured *TADOConnection* component and set the *CommandText* property to a valid query.
3. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

4. Right click in the Fields editor and choose Add Fields.
5. Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open the dataset.

Arranging Persistent Fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields

1. Select the fields. You can select and order one or more fields at a time.
2. Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use Ctrl+Up and Ctrl+Dn to change an individual field's order in the list.

Defining New Persistent Fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor by right clicking and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's FieldName property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's Name property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.
- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

Special persistent field kinds

Field Kind	Purpose
Data	Replaces an existing field (for example to change its data type)
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets)
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in Defining a lookup field.

The following topics describe how to create different field types:

- Defining a Data Field
- Defining a Calculated Field
- Defining a Lookup Field
- Defining an Aggregate Field

Defining a Data Field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a TSmallIntField with a TIntegerField. Because you cannot change a field's data type directly, you must define a new field to replace it.

Warning: Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
5. Select Data in the Field type radio group if it is not already selected.
6. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the **Object Inspector**. For more information about editing field component properties and events, see Setting persistent field properties and events.

Defining a Calculated Field

A calculated field displays values calculated at runtime by a dataset's OnCalcFields event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
4. Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.
5. Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.
6. Place code that calculates values for the field in the OnCalcFields event handler for the dataset. For more information about writing code to calculate field values, see Programming a calculated field.

Note: To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the **Object Inspector**. For more information about editing field component properties and events, see Setting persistent field properties and events.

Programming a Calculated Field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field

1. Select the dataset component from the **Object Inspector** drop-down list.
2. Choose the **Object Inspector** Events page.
3. Double-click the OnCalcFields property to bring up or create a CalcFields procedure for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the **Object Inspector** drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
[Delphi]
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value  + ' ' +
CustomersZip.Value;
```

```
[C++]
CustomersCityStateZip->Value = CustomersCity->Value + AnsiString(", ") +
```

Note: When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's State property and only recomputing the value when *State* is *dsInternalCalc*. See Using internally calculated fields in client datasets for details.

Defining a Lookup Field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note: Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.

6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the LookupCache property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the LookupDataSet is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of LookupKeyFields values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

If every record of *DataSet* has different values for KeyFields, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call RefreshLookupList to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the LookupResultField for every set of LookupKeyFields values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

Defining an Aggregate Field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See Using maintained aggregates for details about maintained aggregates.

To create an aggregate field in the New Field dialog box

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added to the client dataset and its Aggregates property is automatically updated to include the appropriate aggregate specification.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see Specifying aggregates.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

Deleting Persistent Field Components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table.

To remove one or more persistent field components for a dataset

1. Select the field(s) to remove in the Fields editor list box.
2. Press Del.

Note: You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see Creating persistent fields

Note: If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

Setting Persistent Field Properties and Events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a TDBGrid, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the **Object Inspector**.

The following topics discuss using persistent field properties and events:

- Setting Display and Edit Properties at Design Time
- Setting Field Component Properties at Runtime
- Creating Attribute Sets for Field Components
- Controlling and Masking User Input
- Using Default Formatting for Numeric, Date, and Time Fields
- Handling Events

Setting Display and Edit Properties at Design Time

To edit the display properties of a selected field component, switch to the Properties page on the **Object Inspector** window. The following table summarizes display properties that can be edited.

Field component properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. *True*: displays monetary values.False (default): does not display monetary values.

DisplayFormat	Specifies the format of data displayed in a data-aware component.
DisplayLabel	Specifies the column name for a field in a data-aware grid component.
DisplayWidth	Specifies the width, in characters, of a grid column that display this field.
EditFormat	Specifies the edit format of data in a data-aware component.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field to create.
FieldName	Specifies the actual name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the table used to look up field values when *Lookup* is *True*.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the component name of the field component within Delphi.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.
ReadOnly	*True*: Displays field values in data-aware controls, but prevents editing. *False* (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.
Tag	Long integer bucket available for programmer use in every component as needed.
Transliterate	*True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. *False*: Locale translation does not occur.
Visible	*True* (the default): Permits display of field in a data-aware grid.False: Prevents display of field in a data-aware grid component.User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type TStringField does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type TFloatField does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as Calculated, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see Controlling and masking user input.

Setting Field Component Properties at Runtime

You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name.

For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
[Delphi]
CustomersCityStateZip.ReadOnly := True;
```

```
[C++]
CustomersCityStateZip->ReadOnly = true;
```

And this statement changes field ordering by setting the Index property of the *CityStateZip* field in the *Customers* table to 3:

```
[Delphi]
CustomersCityStateZip.Index := 3;
```

```
[C++]
CustomersCityStateZip->Index = 3;
```

Creating Attribute Sets for Field Components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

Note: Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.

To create an attribute set based on a field component in a dataset

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the **Object Inspector**.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Once you have created a new attribute set and added it to the Data Dictionary, you can then associate it with other persistent field components. Even if you later remove the association, the attribute set remains defined in the Data Dictionary.

Associating Attribute Sets with Field Components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Invoke the context menu and choose Associate Attributes.
4. Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Warning: If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor and multi-select field components within a dataset when reapplying attributes.

Removing Attribute Associations

If you change your mind about associating an attribute set with a field, you can remove the association.

To remove an attribute association

1. Invoke the Fields editor for the dataset containing the field.
2. Select the field or fields from which to remove the attribute association.
3. Invoke the context menu for the Fields editor and choose Unassociate Attributes.

Warning: Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the **Object Inspector**.

Controlling and Masking User Input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with TStringField, TDateField, TTimeField, and TDateTimeField, and TSQLTimeStampField components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the **Object Inspector**.

Note: For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component

1. Select the component in the Fields editor or **Object Inspector**.
2. Click the Properties page in the **Object Inspector**.
3. Double-click the values column for the EditMask field in the **Object Inspector**, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

Using Default Formatting for Numeric, Date, and Time Fields

Delphi provides built-in display and edit format routines and intelligent default formatting for TFloatField, TCurrencyField, TBCDField, TFMTBCDField, TIntegerField, TSmallIntField, TWordField, TDateField, TDateTimeField, TTimeField, and TSQLTimeStampField components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Field component formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*,
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField, TBCDField*
BcdToStrF	TFMTBCDField

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the EditFormat is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and EditFormat properties of a field component to override the default display settings for that field. You can also write OnGetText and OnSetText event handlers to do custom formatting for field components at runtime.

Handling Events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

Field component events

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and OnSetText events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. OnValidate is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component

1. Select the component.
2. Select the Events page in the **Object Inspector**.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

Working with Field Component Methods at Runtime

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as BeforePost). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's FocusControl method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
[Delphi]
CustomersCompany.FocusControl;
```

```
[C++]
CustomersCompany->FocusControl();
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see TField.

Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

Displaying, Converting, and Accessing Field Values

Data-aware controls such as TDBEdit and TDBGrid automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Using data controls.

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls,

your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the OnDataChange event lets you know when you may need to update a control's value and the OnStateChange event can help you determine when to enable or disable controls. For more information on these events, see Responding to changes mediated by the data source.

The following topics discuss how to work with field values so that you can display them in standard controls:

- Displaying Field Component Values in Standard Controls
- Converting Field Values
- Accessing Field Values with the Default Dataset Property
- Accessing Field Values with a Dataset's Fields Property
- Accessing Field Values with a Dataset's FieldByName Method

Displaying Field Component Values in Standard Controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a TEdit control because the value of the *CustomersCompany* field may have changed:

```
[Delphi]
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

```
[C++]
void __fastcall TForm1::Table1DataChange(TObject *Sender, TField *Field)
{
  Edit3->Text = CustomersCompany->Value;
}
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in properties for handling conversions.

Note: You can also use Variants to access and set field values.

Converting Field Values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

Field Components	AsVariant	AsString	AsInteger	AsFloat, AsCurrency, AsBCD	AsDateTime, AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		

TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any datatypes not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a TDateTimeField value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. The following table lists permissible conversions that produce special results:

Special conversion results

Conversion	Result
String to Boolean	Converts "True," "False," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime or SQLTimeStamp to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "True" or "False."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
[Delphi]
Edit1.Text := CustomersCustNo.AsString;
```

```
[C++]
Edit1->Text = CustomersCustNo->AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
[Delphi]
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

```
[C++]
MyTableMyField->AsInteger = StrToInt(Edit1->Text);
```

Accessing Field Values with the Default Dataset Property

The most general method for accessing a field's value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
[Delphi]
Customers.FieldValues['CustNo'] := Edit2.Text;
```

```
[C++]
Customers->FieldValues["CustNo"] = Edit2->Text;
```

Because the *FieldValues* property is of type *Variant*, it automatically converts other datatypes into a *Variant* value.

Accessing Field Values with a Dataset's Fields Property

You can access the value of a field with the Fields property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
[Delphi]
Edit1.Text := CustTable.Fields[6].AsString;
```

```
[C++]
Edit1->Text = CustTable->Fields->Fields[6]->AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
[Delphi]
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

```
[C++]
Customers->Edit();
Customers->Insert();
Customers->Fields->Fields[6]->AsString = Edit1->Text;
Customers->Post();
```

Accessing Field Values with a Dataset's FieldByName Method

You can access the value of a field with a dataset's FieldByName method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
[Delphi]
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

```
[C++]
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

Conversely, you can assign a value to a field:

```
[Delphi]
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

```
[C++]
Customers->Edit();
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
Customers->Post();
```

Setting a Default Value for a Field

You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the DefaultExpression property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

Note: If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the dataset posts the record containing the field, before the edited record is applied to the database server.

Working with Constraints

Field components in client datasets or BDE-enabled datasets can use SQL server constraints. In addition, your applications can create and use custom constraints for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.

Creating a Custom Constraint

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Note: Custom constraints are only available in BDE-enabled and client datasets.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

Using Server Constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally.

ImportedConstraint is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.

To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

Using Object Fields

Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields are fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

Types of object field components

Component Name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's ObjectView property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields.

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as TDBEdit that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A TDBGrid control displays ADT and array field data differently, depending on the value of the dataset's ObjectView property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT

or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma-delimited string containing the child fields.

The following topics discuss each type of object field in more detail:

- Working with ADT Fields
- Working with Array Fields
- Working with Dataset Fields
- Working with Reference Fields

Working with ADT Fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called CityEdit, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the Customer table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
[Delphi]
CityEdit.Text := CustomerAddrCity.AsString;
```

```
[C++]
CityEdit->Text = CustomerAddrCity->AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's ObjectView property to True.

Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's FieldByName method by qualifying the name of the child field with the ADT field's name:

```
[Delphi]
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

Using the dateset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
[Delphi]
CityEdit.Text := Customer['Address.City'];
```

```
[C++]
CityEdit->Text = Customer->FieldValues["Address.City"];
```

Note that you can omit the property name (*FieldValues*) because *FieldValues* is the dataset's default property.

Note: Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is False.

Using the ADT field's FieldValues property

You can access the value of a child field with the TADTField's FieldValues property. FieldValues accepts and returns a Variant, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
[Delphi]
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

```
[C++]
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->FieldValues[1];
```

Because FieldValues is the default property of TADTField, the property name (FieldValues) can be omitted. Thus, the following statement is equivalent to the one above:

```
[Delphi]
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Using the ADT field's Fields property

Each ADT field has a Fields property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
[Delphi]
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

```
[C++]
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->Fields->
```

or by name:

```
[Delphi]
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields.FieldByName
('City').AsString;
```

```
[C++]
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->
```

```
[C++]
CityEdit->Text = Customer->FieldByName("Address.City")->AsString;
```

Working with Array Fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The SparseArrays property of *TDataSet* determines whether a unique TField object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's ObjectView property must be set to True before you can access the elements of an array field.

Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the Customer table component represent the *TelNos_Array* field and its six elements:

```
[Delphi]
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

```
[C++]
CustomerTELNOS_ARRAY: TArrayField;
CustomerTELNOS_ARRAY0: TStringField;
CustomerTELNOS_ARRAY1: TStringField;
CustomerTELNOS_ARRAY2: TStringField;
CustomerTELNOS_ARRAY3: TStringField;
CustomerTELNOS_ARRAY4: TStringField;
CustomerTELNOS_ARRAY5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
[Delphi]
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

```
[C++]
TelEdit->Text = CustomerTELNOS_ARRAY0->AsString;
```

Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
[Delphi]
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

```
[C++]
TelEdit->Text = ((TArrayField*)Customer->FieldByName("TelNos_Array"))->FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
[Delphi]
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Using the array field's Fields property

TArrayField has a Fields property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
[Delphi]
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

```
[C++]
for (int i = 0; i < OrderDates->Size; ++i)
  if (!OrderDates->Fields->Fields[i]->IsNull)
    OrderDateListBox->Items->Add(OrderDates->Fields->Fields[i]->AsString);
```

Working with DataSet Fields

Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a TDBGrid control, a dataset field is indicated in each cell of a dataset column with the string "(DataSet)", and at runtime an ellipsis button also exists to the right.

Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's ShowPopupEditor method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
[Delphi]
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

```
[C++]
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a BDE-enabled dataset uses *TNestedTable* to represent the data in its dataset fields, while client datasets use other client datasets.

To access the data in a dataset field

1. Create a persistent TDataSetField object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the values in that dataset field. It must be of a type compatible with the parent dataset.
3. Set that DataSetField property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

Working with Reference Fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the Fields property on the *TReferenceField*.

Displaying reference fields

In a TDBGrid control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's ShowPopupEditor method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
[Delphi]
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

```
[C++]
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset.

To access data in a reference field

1. Create a persistent TDataSetField object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the value of that dataset field.
3. Set that DataSetField property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
[Delphi]
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

```
[C++]
CityEdit->Text = CustomerADDRESS_REF->NestedDataSet->Fields->Fields[1]->AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

To assign a reference field, you need to first use a SELECT statement to select the reference from the table, and then assign. For example:

```
[Delphi]
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

```
[C++]
AddressQuery->SQL->Text = "SELECT REF(A) FROM AddressTable A WHERE A.City = "San Francisco"";
AddressQuery->Open();
CustomerAddressRef->Assign(AddressQuery->Fields->Fields[0]);
```

Using the Borland Database Engine

Using the Borland Database Engine

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your edition of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access) and an ODBC adapter that lets you supply your own ODBC drivers.

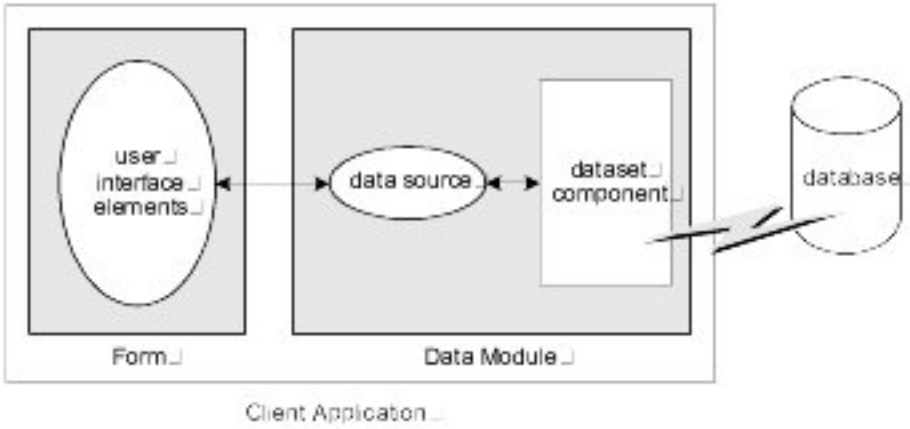
When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broad range of support for database manipulation. Although you can use the BDE's API directly in your application, the components on the BDE category of the **Tool palette** wrap most of this functionality for you.

BDE-based Architecture

When using the BDE, your application uses a variation of the general database architecture described in Database Architecture. In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include

- One or more database components to control transactions and to manage database connections.
- One or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between the components in a BDE-based application are illustrated in the following figure:



The following topics provide additional information about these components:

- Using BDE-enabled Datasets
- Connecting to Databases with TDatabase
- Managing Database Sessions

Using BDE-enabled Datasets

BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Understanding datasets, using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for

- Associating a dataset with database and session connections.
- Caching BLOBs.
- Obtaining a BDE handle.

There are three BDE-enabled datasets:

- *TTable*, a table type dataset that represents all of the rows and columns of a single database table. See Using TTable for a description of features unique to *TTable*.
- *TQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See Using TQuery for a description of features unique to *TQuery*.
- *TStoredProc*, a stored procedure-type dataset that executes a stored procedure that is defined on a database server. See Using TStoredProc for a description of features unique to *TStoredProc*.

Note: In addition to the three types of BDE-enabled datasets, there is a BDE-based client dataset (*TBDEClientDataSet*) that can be used for caching updates.

Associating a Dataset with Database and Session Connections

In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session.

Databases represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a TDatabase component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component for you. Using an explicitly-created *TDatabase* component

is recommended for most applications, because the database component gives you greater control over how the connection is established, including the login process, and lets you create and use transactions.

To associate a BDE-enabled dataset with a database, use the DatabaseName property. DatabaseName is a string that contains different information, depending on whether you are using an explicit database component and, if not, the type of database you are using:

- If you are using an explicit *TDatabase* component, *DatabaseName* is the value of the *DatabaseName* property of the database component.
- If you want to use an implicit database component and the database has a BDE alias, you can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on).
- If you want to use an implicit database component for a Paradox or dBASE database, you can also use *DatabaseName* to simply specify the directory where the database tables are located.

A session provides global management for a group of database connections in an application. When you add BDE-enabled datasets to your application, your application automatically contains a session component, named Session. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. You can control database connections and access to Paradox files using the properties, events, and methods of the session.

You can use the default session to control all database connections in your application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application. To associate your dataset with an explicitly created session component, use the SessionName property. If you do not use explicit session components in your application, you do not have to provide a value for this property. Whether you use the default session or explicitly specify a session using the *SessionName* property, you can access the session associated with a dataset by reading the DBSession property.

Note: If you use a session component, the *SessionName* property of a dataset must match the *SessionName* property for the database component with which the dataset is associated.

Caching BLOBs

BDE-enabled datasets all have a CacheBlobs property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

Working with BDE Handle Properties

You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters.

All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime:

- Handle is a handle to the BDE cursor that accesses the records in the dataset.
- DBHandle is a handle to the database that contains the underlying tables or stored procedure.

- DBLocale is a handle to the BDE language driver for the dataset. The locale controls the sort order and character set used for string data.

These properties are automatically assigned to a dataset when it is connected to a database server through the BDE.

Using TTable

TTable encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality introduced by TDataSet, as well as all of the special features typical of table type datasets.

Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session. Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the TableName property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the TableType property.

Note: The table must be closed when you change its association to a database, session, or database table, or when you set the *TableType* property. However, before you close the table to change these properties, first post or discard any pending changes. If cached updates are enabled, call the ApplyUpdates method to write the posted changes to the database.

TTable components are unique in the support they offer for local database tables (Paradox, dBASE, FoxPro, and comma-delimited ASCII text tables). The following topics describe the special properties and methods that implement this support:

- Specifying the Table Type for Local Tables
- Controlling Read/Write Access to Local Tables
- Specifying a dBASE Index File
- Renaming Local Tables

In addition, *TTable* components can take advantage of the BDE's support for batch operations (table level operations to append, update, delete, or copy entire groups of records). This support is described in Importing data from another table.

Specifying the Table Type for Local Tables

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the TableType property to determine the table's type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its filename extension. The following table summarizes the file extensions recognized by the BDE and the assumptions it makes about a table's type:

Table types recognized by the BDE based on file extension

Extension	Table Type
No file extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII text

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in the previous table, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. The following table indicates the values you can assign to *TableType*:

TableType values

Value	Table Type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

Controlling Read/Write Access to Local Tables

Like any table type dataset, *TTable* lets you control read and write access by your application using the ReadOnly property.

In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The Exclusive property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
[Delphi]
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

```
[C++]
CustomersTable->Exclusive = true; // Set request for exclusive lock
CustomersTable->Active = true; // Now open the table
```

Note: You can attempt to set *Exclusive* on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

Specifying a dBASE Index File

For most servers, you use the methods common to all table type datasets to specify an index. These methods are described in Sorting records with indexes.

For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (*.NDX), however, you must use the IndexFiles *and* IndexNameproperties instead. Set the *IndexFiles* property to the name of the non-production index file or list the .NDX files. Then, specify one index in the *IndexName* property to have it actively sorting the dataset.

At design time, click the ellipsis button in the *IndexFiles* property value in the **Object Inspector** to invoke the Index Files editor. To add one non-production index file or .NDX file: click the Add button in the Index Files dialog and select the file from the Open dialog. Repeat this process once for each non-production index file or .NDX file. Click the OK button in the Index Files dialog after adding all desired indexes.

This same operation can be performed programmatically at runtime. To do this, access the *IndexFiles* property using properties and methods of string lists. When adding a new set of indexes, first call the *Clear* method of the table's

IndexFiles property to remove any existing entries. Call the *Add* method to add each non-production index file or .NDX file:

```
[Delphi]
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

```
[C++]
Table2->IndexFiles->Clear();
Table2->IndexFiles->Add("Bystate.ndx");
Table2->IndexFiles->Add("Byzip.ndx");
Table2->IndexFiles->Add("Fullname.ndx");
Table2->IndexFiles->Add("St_name.ndx");
```

After adding any desired non-production or .NDX index files, the names of individual indexes in the index file are available, and can be assigned to the *IndexName* property. The index tags are also listed when using the *GetIndexNames* method and when inspecting index definitions through the *TIndexDef* objects in the *IndexDefs* property. Properly listed .NDX files are automatically updated as data is added, changed, or deleted in the table (regardless of whether a given index is used in the IndexName property).

In the example below, the *IndexFiles* for the *AnimalsTable* table component is set to the non-production index file ANIMALS.MDX, and then its *IndexName* property is set to the index tag called "NAME":

```
[Delphi]
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

```
[C++]
AnimalsTable->IndexFiles->Add("ANIMALS.MDX");
AnimalsTable->IndexName = "NAME";
```

Once you have specified the index file, using non-production or .NDX indexes works the same as any other index. Specifying an index name sorts the data in the table and makes it available for indexed-based searches, ranges, and (for non-production indexes) master-detail linking. See Using table type datasets for details on these uses of indexes.

There are two special considerations when using dBASE III PLUS-style .NDX indexes with *TTable* components. The first is that .NDX files cannot be used as the basis for master-detail links. The second is that when activating a .NDX index with the *IndexName* property, you must include the .NDX extension in the property value as part of the index name:

```
[Delphi]
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

```
[C++]
Table1->IndexName = "ByState.NDX";
```

```
TVarRec vr = ("NE");
Table1->FindKey(&vr, 0);
```

Renaming a Table

To rename a Paradox or dBASE table at runtime, call the table's RenameTable method. For example, the following statement renames the Customer table to CustInfo:

```
[Delphi]
Customer.RenameTable('CustInfo');
```

```
[C++]
Customer->RenameTable("CustInfo");
```

Importing Data from Another Table

You can use a table component's BatchMovemethod to import data from another table. *BatchMove* can

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

BatchMove takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. The following table describes the possible settings for the mode specification:

BatchMove import modes

Value	Meaning
batAppend	Append all records from the source table to the end of this table.
batAppendUpdate	Append all records from the source table to the end of this table and update existing records in this table with matching records from the source table.
batCopy	Copy all records from the source table into this table.
batDelete	Delete all records in this table that also appear in the source table.
batUpdate	Update existing records in this table with matching records from the source table.

For example, the following code updates all records in the current table with records from the *Customer* table that have the same values for fields in the current index:

```
[Delphi]
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

```
[C++]
Table1->BatchMove("CUSTOMER.DB", batUpdate);
```

BatchMove returns the number of records it imports successfully.

Warning: Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

BatchMove performs only some of the batch operations supported by the BDE. Additional functions are available using the *TBatchMove* component. If you need to move a large amount of data between or among tables, use *TBatchMove* instead of calling a table's *BatchMove* method. For information about using *TBatchMove*, see Using TBatchMove

Using TQuery

TQuery represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by TDataSet, as well as all of the special features typical of query-type datasets.

Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session. (The one exception is when you use the *TQuery* for a heterogeneous query.) You specify the SQL statement for the query by setting the SQL property.

A TQuery component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase. You must install the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

The following topics discuss features that are unique to *TQuery* components (as opposed to other query-type datasets):

- Creating Heterogeneous Queries.
- Obtaining an Editable Result Set
- Updating Read-only Result Sets

Creating Heterogenous Queries

TQuery supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table. When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query

1. Define separate BDE aliases for each database accessed in the query using the BDE Administration tool or the SQL explorer.
2. Leave the *DatabaseName* property of the *TQuery* blank; the names of the databases used will be specified in the SQL statement.

3 In the SQL property, specify the SQL statement to execute. Precede each table name in the statement with the BDE alias for the table's database, enclosed in colons. This whole reference is then enclosed in quotation marks.

4 Set any parameters for the query in the *Params* property.

5 Call *Prepare* to prepare the query for execution prior to executing it for the first time.

6 Call *Open* or *ExecSQL* depending on the type of query you are executing.

For example, suppose you define an alias called Oracle1 for an Oracle database that has a CUSTOMER table, and Sybase1 for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

Obtaining an Editable Result Set

To request a result set that users can edit in data-aware controls, set a query component's RequestLive property to True. Setting *RequestLive* to True does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser.

- Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. When using Local SQL, a live result set for a query against a single table or view is returned if the query does not contain any of the following:
- DISTINCT in the SELECT clause
- Joins (inner, outer, or UNION)
- Aggregate functions with or without GROUP BY or HAVING clauses
- Base tables or views that are not updatable
- Subqueries
- ORDER BY clauses not based on an index
- Queries against a remote database server are parsed by the server. If the *RequestLive* property is set to True, the SQL statement must abide by Local SQL standards in addition to any server-imposed restrictions because the BDE needs to use it for conveying data changes to the table. A live result set for a query against a single table or view is returned if the query does not contain any of the following:
- A DISTINCT clause in the SELECT statement
- Aggregate functions, with or without GROUP BY or HAVING clauses
- References to more than one base table or updatable views (joins)
- Subqueries that reference the table in the FROM clause or other tables

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *True*. Even if the query returns a live result set, you may not be able to update the result set directly if it contains

linked fields or you switch indexes before attempting an update. If these conditions exist, you should treat the result set as a read-only result set, and update it accordingly.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for SQL queries made against a remote server.

Updating a Read-only Result Set

Applications can update data returned in a read-only result set if they are using cached updates.

If you are using a client dataset to cache updates, the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates:

If all updates are applied to a single database table, you can indicate the underlying table to update in an OnGetTableName event handler.

If you need more control over applying updates, you can associate the query with an update object (TUpdateSQL). A provider automatically uses this update object to apply updates:

- Associate the update object with the query by setting the query's UpdateObject property to the *TUpdateSQL* object you are using.
- Set the update object's ModifySQL, InsertSQL, and DeleteSQL properties to SQL statements that perform the appropriate updates for your query's data.

You must use an update object if you are using the BDE to cache updates.

Note: For more information on using update objects, see Using update objects to update a dataset.

Using TStoredProc

TStoredProc represents a stored procedure. It implements all of the basic functionality introduced by TDataSet, as well as most of the special features typical of stored procedure-type datasets.

Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session. Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the StoredProcName property.

TStoredProc differs from other stored procedure-type datasets in the following ways:

- It gives you greater control over how to bind parameters.
- It provides support for Oracle overloaded stored procedures.

Binding Parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

TStoredProc lets you use the ParamBindMode property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is

significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

Tip: If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order.

Working with Oracle Overloaded Stored Procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's Overload property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

Note: Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

Connecting to Databases with TDatabase

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a TDatabase component. A database component represents the connection to a single database in the context of a BDE session.

TDatabase performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Connecting to databases.

In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics:

- Associating a Database Component with a Session
- Understanding Database and Session Component Interactions
- Identifying the Database
- Opening a Connection Using TDatabase
- Using Database Components in Data Modules
- Applying Cached Updates Using a Database.

Associating a Database Component with a Session

All database components must be associated with a BDE session. Use the SessionName, establish this association. When you first create a database component at design time, *SessionName* is set to "Default", meaning that it is associated with the default session component that is referenced by the global Session variable.

Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add TSession components for each session. Then, associate your dataset with a session component by setting the SessionName property to a session component's SessionName property.

At runtime, you can access the session component with which the database is associated by reading the Session property. If *SessionName* is blank or "Default", then the *Session* property references the same *TSession* instance

referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component's parent session component without knowing the session's actual name.

For more information about BDE sessions, see Managing database sessions.

If you are using an implicit database component, the session for that database component is the one specified by the dataset's SessionName property.

Understanding Database and Session Component Interactions

In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session's KeepConnections property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default OnPasswordevent for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.

Session methods apply somewhat differently. TSession methods affect all database components, regardless of whether they are explicitly created or instantiated implicitly by a dataset. For example, the session method DropConnectionscloses all datasets belonging to a session's database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *True*.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1.CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

Identifying the Database

AliasName and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects.

AliasName specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

You create and edit BDE aliases using the Database Explorer . For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

DriverNameis the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

DatabaseName lets you provide your own name for a database connection. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. *DatabaseName* can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to let you link them to database components.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

Note: The Database Properties editor also lets you view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. For information on connection parameters, see Setting BDE Alias Parameters. For information on *LoginPrompt*, see Controlling Server Login. For information on *KeepConnection* see Opening a Connection Using TDatabase.

Setting BDE Alias Parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the **Object Inspector** to invoke the String List editor.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the Params property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly.

Identifying the Database

As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in Connecting to a database server. Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component's KeepConnection property is *True*.

When you connect to a remote database server from an application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can also communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the Params property of a database component. For information about SQL Links parameters, see the online *SQL Links* User's Guide. To edit the *Params* property, see Setting BDE alias parameters

Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- Are the DLLs for your connection and database drivers in the search path?
- If you are using TCP/IP:
- Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?

- Is the server's IP address registered in the client's HOSTS file?
- Is the Domain Name Services (DNS) properly configured?
- Can you ping the server?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.

Using Database Components in Data Modules

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the HandleShared property of the database component to *True* to prevent global name space conflicts.

Managing Database Sessions

An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

All BDE-based database applications automatically include a default session component, named Session, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are implicit (created by the session at runtime when you open a dataset that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

To use the default session, you need write no code unless your application must

- Explicitly activate or deactivate a session, enabling or disabling the session's databases' ability to open.
- Modify the properties of the session, such as specifying default properties for implicitly generated database components.
- Execute a session's methods, such as managing database connections (for example opening and closing database connections in response to user actions).
- Respond to session events, such as when the application attempts to access a password-protected Paradox or dBASE table.
- Set Paradox directory locations such as the *NetFileDir* property to access Paradox tables on a network and the *PrivateDir* property to a local hard drive to speed performance.
- Manage the BDE aliases that describe possible database connection configurations for databases and datasets that use the session.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session. If you open a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.

- Associates the database component with the default session.
- Initializes some of the database component's key properties based on the default session's properties. Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application.

The default session provides a widely applicable set of defaults that can be used as is by most applications. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions, where each thread has its own session.

Applications can create additional session components as needed. BDE-based database applications automatically include a session list component, named Sessions, that you can use to manage all of your session components. For more information about managing multiple sessions see, Managing multiple sessions.

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

Activating a Session

Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets). Setting *Active* to *True* triggers a session's OnStartup event, registers the paradox directory locations. with the BDE, and registers the *ConfigMode* property, which determines what BDE aliases are available within the session. You can write an *OnStartup* event handler to initialize the *NetFileDir*, *PrivateDir*, and *ConfigMode* properties before they are registered with the BDE, or to perform other specific session start-up activities.

Once a session is active, you can open its database connections by calling the OpenDatabase method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may trigger events associated with them.

Note: You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
[Delphi]
Session1.Close;
```

```
[C++]
Session1->Close();
```

This code sets Session1's *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
[Delphi]
Session1.Open;
```

```
[C++]
Session1->Open();
```

Note: If a session is active you can also open and close individual database connections. For more information, see Closing database connections.

Specifying Default Database Connection Behavior

KeepConnections provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

Note: Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see Managing database connections.

KeepConnections should be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

Note: Even when *KeepConnections* is *True* for a session, you can close and free inactive database connections for all implicit database components by calling the DropConnections method. For more information about *DropConnections*, see Dropping inactive database connections.

Managing Database Connections

You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to

- Open database connections.
- Close database connections.
- Close and free all inactive temporary database connections.
- Locate specific database connections.
- Iterate through all open database connections.

Opening Database Connections

To open a database connection within a session, call the OpenDatabase method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
[Delphi]
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

```
[C++]
TDatabase *BCDemosDatabase = Session->OpenDatabase("BCDEMOS");
```

OpenDatabase actives the session if it is not already active, and then checks if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open.

Closing Database Connections

To close an individual database connection, call the CloseDatabase method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.

```
[Delphi]
Session.CloseDatabase(DBDemosDatabase);
```

```
[C++]
Session->CloseDatabase(BCDemosDatabase);
```

If the specified database name is associated with a temporary (implicit) database component, and the session's *KeepConnections* property is *False*, the database component is freed, effectively closing the connection.

Note: If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

Note: Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

There are two ways to close all database connections within the session:

- Set the *Active* property for the session to *False*.
- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

Dropping Inactive Database Connections

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the DropConnections method. For example, the following code frees all inactive, temporary database components for the default session:

```
[Delphi]
Session.DropConnections;
```

```
[C++]
Session->DropConnections();
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call Close.

Searching for a Database Connection

Use a session's FindDatabase method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name.

FindDatabase returns the database component if it finds a match. Otherwise it returns **nil**.

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
[Delphi]
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then                          { database doesn't exist for session so,}
    DB := Session.OpenDatabase('DBDEMOS');    { create and open it}
  if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
  end;
end;
```

```
[C++]
TDatabase *DB = Session->FindDatabase("BCDEMOS");
if ( !DB )                                  // Database does not exist for session so
  DB = Session->OpenDatabase("BCDEMOS"); // create and open it
if (DB && DB->Connected)
{
  if (!DB->InTransaction)
  {
    DB->StartTransaction();
    .
    .
    .
  }
}
```

Iterating Through a Session's Database Components

You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session.

Databases is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
[Delphi]
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
```

```
[C++]
if (Session->DatabaseCount > 0)
  for (int MaxDbCount = 0; MaxDbCount < Session->DatabaseCount; MaxDbCount++)
    Session->Databases[MaxDbCount]->KeepConnection = true;
```

Working with Password-protected Paradox and dBASE Tables

A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.

Using the AddPassword method

The AddPassword method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBASE table that requires a password for access. If you do not add the password to the session, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

AddPassword takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords (one at a time) to access tables protected with different passwords.

```
[Delphi]
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
```

```
    Application.Terminate;
  end;
end;
```

```
[C++]
AnsiString PassWrd;
PassWrd = InputBox("Enter password", "Password:", "");
Session->AddPassword(PassWrd);
try
{
  Table1->Open();
}
catch(...)
{
  ShowMessage("Could not open table!");
  Application->Terminate();
}
```

Note: Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form.

The Add button of the PasswordDialog function dialog has the same effect as the *AddPassword* method.

```
[Delphi]
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

```
[C++]
if (PasswordDlg(Session))
  Table1->Open();
else
  ShowMessage("No password given, could not open table!");
```

Using the RemovePassword and RemoveAllPasswords methods

RemovePassword deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
[Delphi]
Session.RemovePassword('secret');
```

```
[C++]
Session->RemovePassword("secret");
```

RemoveAllPasswords deletes all previously added passwords from memory.

```
[Delphi]
Session.RemoveAllPasswords;
```

```
[C++]
Session->RemoveAllPasswords();
```

Using the GetPassword method and OnPassword event

The OnPassword event allows you to control how your application supplies passwords for Paradox and dBASE tables when they are required. Provide a handler for the *OnPassword* event if you want to override the default password handling behavior. If you do not provide a handler, Delphi presents a default dialog for entering a password and no special behavior is provided—the table open attempt either succeeds or an exception is raised.

If you provide a handler for the *OnPassword* event, do two things in the event handler: call the *AddPassword* method and set the event handler's *Continue* parameter to True. The *AddPassword* method passes a string to the session to be used as a password for the table. The *Continue* parameter indicates to Delphi that no further password prompting need be done for this table open attempt. The default value for *Continue* is *False*, and so requires explicitly setting it to True. If *Continue* is *False* after the event handler has finished executing, an *OnPassword* event fires again—even if a valid password has been passed using *AddPassword*. If Continue is True after execution of the event handler and the string passed with *AddPassword* is not the valid password, the table open attempt fails and an exception is raised.

OnPassword can be triggered by two circumstances. The first is an attempt to open a password-protected table (dBASE or Paradox) when a valid password has not already been supplied to the session. (If a valid password for that table has already been supplied, the OnPassword event does not occur.)

The other circumstance is a call to the GetPassword *method. GetPassword either generates an* OnPassword event, or, if the session does not have an OnPassword event handler, displays a default password dialog. It returns True *if the OnPassword event handler or default* dialog added a password to the session, and False *if no entry at all was made.*

In the following example, the *Password* method is designated as the *OnPassword* event handler for the default sessionby assigning it to the global *Session* object's *OnPassword* property.

```
[Delphi]
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

```
[C++]
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->OnPassword = Password;
}
```

In the *Password* method, the *InputBox* function prompts the user for a password. The *AddPassword* method then programmatically supplies the password entered in the dialog to the session.

```
[Delphi]
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrd > '');
  Session.AddPassword(Passwrd);
end;
```

```
[C++]
void __fastcall TForm1::Password(TObject *Sender, bool &Continue)
{
  AnsiString PassWrd = InputBox("Enter password", "Password:", "");
  Session->AddPassword(PassWrd);
  Continue = (PassWrd > "");
}
```

The *OnPassword* event (and thus the *Password* event handler) is triggered by an attempt to open a password-protected table, as demonstrated below. Even though the user is prompted for a password in the handler for the *OnPassword* event, the table open attempt can still fail if they enter an invalid password or something else goes wrong.

```
[Delphi]
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
CRLF = #13 + #10;
begin
  try
    Table1.Open;                              { this line triggers the OnPassword event }
  except
    on E:Exception do begin                          { exception if cannot open table }
      ShowMessage('Error!' + CRLF +           { display error explaining what happened }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;                                  { end the application }
    end;
  end;
end;
```

```
[C++]
void __fastcall TForm1::OpenTableBtnClick(TObject *Sender)
{
  try
  {
    // this line triggers the OnPassword event
    Table1->Open();
  }
  // exception if cannot open table
  catch(...)
  {
    ShowMessage("Could not open table!");
    Application->Terminate();
  }
}
```

Specifying Paradox Directory Locations

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables.

NetFileDir specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the **Object Inspector**. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
[Delphi]
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

```
[C++]
Session->NetFileDir = ExtractFilePath(ParamStr(0));
```

Note: *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements. If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

Note: Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
[Delphi]
Session.PrivateDir := "C:\TEMP";
```

```
[C++]
Session->PrivateDir = "C:\\TEMP";
```

Warning: Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

Working with BDE Aliases

Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime.

The AddAlias method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
[Delphi]
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

```
[C++]
TStringList *AliasParams = new TStringList();
try
{
  AliasParams->Add("OPEN MODE=READ");
  AliasParams->Add("USER NAME=TOMSTOPPARD");
  AliasParams->Add("SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB");
  Session->AddAlias("CATS", "INTRBASE", AliasParams);
  .
  .
  .
}
catch (...)
{
  delete AliasParams;
  throw;
}
delete AliasParams;
```

AddStandardAlias creates a new BDE alias for Paradox, dBASE, or ASCII tables. *AddStandardAlias* takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For example, the following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
[Delphi]
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

```
[C++]
Session->AddStandardAlias("MYBCDEMOS", "C:\\TESTING\\DEMOS\\", "Paradox");
```

When you add an alias to a session, the BDE stores a copy of the alias in memory, where it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. *ConfigMode* is a set that describes which types of aliases can be used by the databases in the session. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent, cfmSession*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session (*cfmSession*), all aliases in the BDE configuration file on a user's system (*cfmPersistent*), and all aliases that the BDE maintains in memory (*cfmVirtual*). You can change *ConfigMode* to restrict what BDE aliases the databases in a session can use. For example, setting *ConfigMode* to *cfmSession*

restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

To make a newly created alias available to all sessions and to other applications, use the session's SaveConfigFile method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

After you create an alias, you can make changes to its parameters by calling ModifyAlias. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values. For example, the following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
[Delphi]
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

```
[C++]
TStringList *List = new TStringList();
List->Clear();
List->Add("OPEN MODE=READ/WRITE");
Session->ModifyAlias("CATS", List);
delete List;
```

To delete an alias previously created in a session, call the DeleteAlias method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

Note: *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Session components provide five methods for retrieving information about a BDE aliases, including parameter information and driver information. They are:

- GetAliasNames, to list the aliases to which a session has access.
- GetAliasParams, to list the parameters for a specified alias.
- GetAliasDriverName, to return the name of the BDE driver used by the alias.
- GetDriverNames, to return a list of all BDE drivers available to the session.
- GetDriverParams, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see Using transactions with the BDE.. For more information about BDE aliases see the BDE online help, BDE32.HLP.

Retrieving Information About a Session

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method

retrieves the names of tables associated with a specific database component used by the session. The following table summarizes the informational methods to a session component:

Database-related informational methods for session components

Method	Purpose
GetAliasDriverName	Retrieves the BDE driver for a specified alias of a database.
GetAliasNames	Retrieves the list of BDE aliases for a database.
GetAliasParams	Retrieves the list of parameters for a specified BDE alias of a database.
GetConfigParams	Retrieves configuration information from the BDE configuration file.
GetDatabaseNames	Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use.
GetDriverNames	Retrieves the names of all currently installed BDE drivers.
GetDriverParams	Retrieves the list of parameters for a specified BDE driver.
GetStoredProcNames	Retrieves the names of all stored procedures for a specified database.
GetTableNames	Retrieves the names of all tables matching a specified pattern for a specified database.
GetFieldNames	Retrieves the names of all fields in a specified table in a specified database.

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
[Delphi]
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

```
[C++]
TStringList *List = new TStringList();
try
{
  Session->GetDatabaseNames(List);
  .
  .
  .
}
catch (...)
{
delete List;
throw;
}
delete List;
```

Creating Additional Sessions

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the **Object Inspector**, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime.

Note: Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime

1. Declare a *TSession* variable.
2. Instantiate a new session by calling the *Create* method. The constructor sets up an empty list of database components for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
3. Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see Naming a session.
4. Activate the session and optionally adjust its properties.

You can also create and open sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For information about *OpenSession*, see Managing multiple sessions..

Naming a Session

A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default," For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to "InterBaseSession," activate the session, and associate an existing database component *Database1* with that session:

```
[Delphi]
var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

```
[C++]
TSession *IBSession = Sessions->OpenSession("InterBaseSession");
Database1->SessionName = "InterBaseSession";
```

For more information about using *Sessions*, see Managing Multiple Sessions..

Managing Multiple Sessions

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE category on the **Tool palette** contains a session component that you can place in a data module or on a form at design time.

Warning: When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the OpenSession method of the global Sessions object at runtime.

OpenSession requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
[Delphi]
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

```
[C++]
Sessions->OpenSession("RunTimeSession" + IntToStr(Sessions->Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

Sessions is a variable of type *TSessionList* that is automatically instantiated for BDE-based database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the *TSessionList* component:

TSessionList properties and methods

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the session list.
FindSession	Searches for a session with a specified name and returns a pointer to it, or **nil** if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*.
GetSessionNames	Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, "Default" for the default session.
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the AutoSessionName property to make sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

Using Transactions with the BDE

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the database component to control transactions. The main advantage to using the methods and properties of a database component is that it provides a clean, portable application that is not dependent on a particular database or server. This type of transaction control is supported by all database connection components, and described in Managing transactions.
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

When working with local databases, you can only use the database component to create explicit transactions (local databases do not support passthrough SQL). However, there are limitations to using local transactions. For more information on using local transactions, see Using Local Transactions.

Note: You can minimize the number of transactions you need by caching updates. For more information about cached updates, see Using a Client Dataset to Cache Updates.

Using Passthrough SQL

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the "Typical" installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol. See your network administrator for more information.
- Have access to a database on a remote server.
- Set SQLPASSTHRU MODE to NOT SHARED using the SQL Explorer. SQLPASSTHRU MODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases,

SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. However, you can't share database connections when using transaction control statements.

Note: When SQLPASSTHRU MODE is NOT SHARED, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

Using Local Transactions

The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

Note: When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- Transactions cannot be run against temporary tables.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.
- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
- Several tables are open.
- The cursor is closed on a table to which no changes were made.

Using the BDE to Cache Updates

The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in Using a client dataset to cache updates.

For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client dataset to cache updates. The following table lists these properties, events, and methods and the corresponding properties, methods and events on *TBDEClientDataSet*:

Properties, methods, and events for cached updates

On BDE-enabled Datasets (or TDatabase)	On TBDEClientDataSet	Purpose
CachedUpdates	Not needed for client datasets, which always cache updates.	Determines whether cached updates are in effect for the dataset.
UpdateObject	Use a *BeforeUpdateRecord* event handler, or, if using *TClientDataSet*, use the *UpdateObject* property on the BDE-enabled source dataset.	Specifies the update object for updating read-only datasets.
UpdatesPending	ChangeCount	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	StatusFilter	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	UpdateStatus	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	OnReconcileError	An event for handling update errors on a record-by-record basis.
OnUpdateRecord	BeforeUpdateRecord	An event for processing updates on a record-by-record basis.
ApplyUpdates (database)	ApplyUpdates	Applies records in the local cache to the database.
CancelUpdates	CancelUpdates	Removes all pending updates from the local cache without applying them.
CommitUpdates	Reconcile	Clears the update cache following successful application of updates.
FetchAll	GetNextPacket (and PacketRecords)	Copies database records to the local cache for editing and updating.
RevertRecord	RevertRecord	Undoes updates to the current record if updates are not yet applied.

For an overview of the cached update process, see Overview of using cached updates.

The following topics describe in more detail on how to use the BDE to cache updates:

- Enabling BDE-based Cached Updates.
- Applying BDE-based Cached Updates.
- Using Update Objects to Update a Dataset.

Note: Even if you are using a client dataset to cache updates, you may want to read the section about update objects. You can use update objects in the *BeforeUpdateRecord* event handler of *TBDEClientDataSet* or *TDataSetProvider* to apply updates from stored procedures or multi-table queries.

Enabling BDE-based Cached Updates

To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the CachedUpdates property to *True*. When you enable cached updates, a copy of all records

is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

The dataset caches all updates until you set *CachedUpdates* to *False*. Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory. Canceling the updates by calling CancelUpdates removes all the changes currently in the cache, but does not stop the dataset from caching any subsequent changes.

Note: If you disable cached updates by setting *CachedUpdates* to *False*, any pending changes that you have not yet applied are discarded without notification. To prevent losing changes, test the UpdatesPending property before disabling cached updates.

Applying BDE-based Cached Updates

Applying updates is a two-phase process that should occur in the context of a database component's transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see Managing Transactions.

When applying updates under database transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.
3. The transaction is committed if writes are successful or rolled back if they are not,

Write Status	Transaction
Successful	Database changes are committed, ending the database transaction.
	Cached updates are committed, clearing the internal cache buffer (phase 2).
Unsuccessful	Database changes are rolled back, ending the database transaction.
	Cached updates are not committed, remaining intact in the internal cache.

For information about creating and using an *OnUpdateRecord* event handler, see Creating an OnUpdateRecord Event Handler. For information about handling update errors that occur when applying cached updates, see Handling Cached Update errors.

Note: Applying cached updates is particularly tricky when you are working with multiple datasets linked in a master/detail relationship because the order in which you apply updates to each dataset is significant. Usually, you must update master tables before detail tables, except when handling deleted records, where this order must be reversed. Because of this difficulty, it is strongly recommended that you use client datasets when caching updates in a master/detail form. Client datasets automatically handle all ordering issues with master/detail relationships.

There are two ways to apply BDE-based updates:

- You can apply updates using a database component by calling its ApplyUpdates method. This method is the simplest approach, because the database handles all details of managing a transaction for the update process and of clearing the dataset's cache when updating is complete.
- You can apply updates for a single dataset by calling the dataset's ApplyUpdates and CommitUpdates methods. When applying updates at the dataset level you must explicitly code the transaction that wraps the update process as well as explicitly call *CommitUpdates* to commit updates from the cache.

Warning: To apply updates from a stored procedure or an SQL query that does not return a live result set, you must use *TUpdateSQL* to specify how to perform updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. See Using update objects to update a dataset for details.

Applying Cached Updates Using a Database

To apply cached updates to one or more datasets in the context of a database connection, call the database component's ApplyUpdates method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
[Delphi]
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

```
[C++]
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
    Database1->TransIsolation = tiDirtyRead;
  Database1->ApplyUpdates(&CustomersQuery,0);
}
```

The above sequence writes cached updates to the database in the context of an automatically-generated transaction. If successful, it commits the transaction and then commits the cached updates. If unsuccessful, it rolls back the transaction and leaves the update cache unchanged. In this latter case, you should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see Handling cached update errors.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries:

```
[Delphi]
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

```
[C++]
TDBDataSet* ds[] = {CustomerQuery, OrdersQuery};
if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
  Database1->TransIsolation = tiDirtyRead;
Database1->ApplyUpdates(ds,1);
```

Applying Cached Updates with Dataset Component Methods

You can apply updates for individual BDE-enabled datasets directly using the dataset's ApplyUpdates and CommitUpdates methods. Each of these methods encapsulate one phase of the update process:

1. *ApplyUpdates* writes cached changes to a database (phase 1).
2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:

```
[Delphi]
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates;                { try to write the updates to the database }
    Database1.Commit;                                  { on success, commit the changes }
  except
    Database1.Rollback;                                  { on failure, undo any changes }
    raise;                 { raise the exception again to prevent a call to CommitUpdates }
  end;
  CustomerQuery.CommitUpdates;                     { on success, clear the internal cache }
end;
```

```
[C++]
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  Database1->StartTransaction();
  try
  {
    if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
      Database1->TransIsolation = tiDirtyRead;
    CustomerQuery->ApplyUpdates(); // try to write the updates to the database
    Database1->Commit(); // on success, commit the changes
  }
  catch (...)
  {
    Database1->Rollback(); // on failure, undo any changes
    throw; // throw the exception again to prevent a call to CommitUpdates
  }
  CustomerQuery->CommitUpdates(); // on success, clear the internal cache
}
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The raise statement inside the **try...except** block reraises the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

Creating an OnUpdateRecord Event Handler

When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter

substitution, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
[Delphi]
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

```
[C++]
void __fastcall TForm1::DataSetUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Perform updates here...
}
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update that needs to be performed for the current record. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. If you are using an update object, you need to pass this parameter to the update object when applying the update. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update.

The *UpdateAction* parameter indicates whether you applied the update. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. If your event handler successfully applies the update, change this parameter to *uaApplied* before exiting. If you decide not to update the current record, change the value to *uaSkip* to preserve unapplied changes in the cache. If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted and an exception is raised. You can suppress the error message (raising a silent exception) by changing *UpdateAction* to *uaAbort*.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. *OldValue* gives the original field value that was fetched from the database. It can be useful in locating the database record to update. *NewValue* is the edited value in the update you are trying to apply.

Warning: An *OnUpdateRecord* event handler, like an *OnUpdateError* or *OnCalcFields* event handler, should never call any methods that change the current record in a dataset.

The following example illustrates how to use these parameters and properties. It uses a *TTable* component named *UpdateTable* to apply updates. In practice, it is easier to use an update object, but using a table illustrates the possibilities more clearly.

```
[Delphi]
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
```

```
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
        UpdateAction := uaApplied;
end;
```

```
[C++]
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  if (UpdateKind == ukInsert)
  {
    TVarRec values[2];
    for (int i = 0; i < 2; i++)
      values[i] = DataSet->Fields->Fields[i]->NewValue;
    UpdateTable->AppendRecord(values, 1);
  }
  else
  {
    TLocateOptions lo;
    lo.Clear();
    if (UpdateTable->Locate("KeyField", DataSet->Fields->Fields[0]->OldValue, lo))
      switch (UpdateKind)
      {
        case ukModify:
          UpdateTable->Edit();
          UpdateTable->Fields->Fields[1]->Value = DataSet->Fields->Fields[1]->Value;
          UpdateTable->Post();
          break;
        case ukDelete:
          UpdateTable->Delete();
          break;
      }
  }
  UpdateAction = uaApplied;
}
```

Handling Cached Update Errors

The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's OnUpdateError event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Here is the skeleton code for an *OnUpdateError* event handler:

```
[Delphi]
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
```

```
  { ... perform update error handling here ... }
end;
```

```
[C++]
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Respond to errors here...
}
```

DataSet references the dataset to which updates are applied. You can use this dataset to access new and old values during error handling. The original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. These values provide the only way to inspect and change update values in the event handler.

Warning: Do not call any dataset methods that change the current record (such as *Next* and *Prior*). Doing so causes the event handler to enter an endless loop.

The *E* parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
[Delphi]
ErrorLabel.Caption := E.Message;
```

```
[C++]
ErrorLabel->Caption = E->Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it.

The *UpdateKind* parameter describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

UpdateKind values

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting a new record caused an error.
ukDelete	Deleting an existing record caused an error.

UpdateAction tells the BDE how to proceed with the update process when your event handler exits. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler:

- If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.
- When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.

- Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

The following code shows an *OnUpdateError* event handler that checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
[Delphi]
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip                    { key violation, just skip this record }
    else
      UpdateAction := uaAbort;              { don't know what's wrong, abort the update }
  end;
```

```
[C++]
// include BDE.hpp in your unit file for this example
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  UpdateAction = uaFail // initialize to fail the update
  if (E->ClassNameIs("EDBEngineError"))
  {
    EDBEngineError *pDBE = (EDBEngineError *)E;
    if (pDBE->Errors[pDBE->ErrorCount - 1]->ErrorCode == DBIERR_KEYVIOL)
      UpdateAction = uaSkip; // Key violation, just skip this record
  }
}
```

Note: If an error occurs during the application of cached updates, an exception is raised and an error message displayed. Unless the *ApplyUpdates* is called from within a try...except construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

Using Update Objects to Update a Dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not "live", it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object.

To update a dataset

1. If you are using a client dataset, use an external provider component with *TClientDataSet* rather than *TBDEClientDataSet*. This is so you can set the *UpdateObject* property of the BDE-enabled source dataset (step 3).
2. Add a TUpdateSQL component to the same data module as the BDE-enabled dataset.
3. Set the BDE-enabled dataset component's UpdateObject property to the *TUpdateSQL* component in the data module.
4. Specify the SQL statements needed to perform updates using the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. You can use the Update SQL editor to help you compose these statements.

5 Close the dataset.

6 Set the dataset component's *CachedUpdates* property to *True* or link the dataset to the client dataset using a dataset provider.

7 Reopen the dataset.

Note: Sometimes, you need to use multiple update objects. For example, when updating a multi-table join or a stored procedure that represents data from multiple datasets, you must provide one *TUpdateSQL* object for each table you want to update. When using multiple update objects, you can't simply associate the update object with the dataset by setting the *UpdateObject* property. Instead, you must manually call the update object from an OnUpdateRecord event handler (when using the BDE to cache updates) or a BeforeUpdateRecord event handler (when using a client dataset).

The update object actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- ModifySQL specifies the UPDATE statement.
- InsertSQL specifies the INSERT statement.
- DeleteSQL specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

1 Selects an SQL statement to execute based on whether the current record is modified, inserted, or deleted.

2 Provides parameter values to the SQL statement.

3 Prepares and executes the SQL statement to perform the specified update.

Creating SQL Statements for Update Components

To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table.

The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:

```
[Delphi]
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

```
[C++]
UpdateSQL->DeleteSQL->Clear();
UpdateSQL->DeleteSQL->Add("DELETE FROM Inventory I");
UpdateSQL->DeleteSQL->Add("WHERE (I.ItemNo = :OLD_ItemNo)");
```

At design time, you can use the Update SQL editor to help you compose the SQL statements that apply updates.

Update objects provide automatic parameter binding for parameters that reference the dataset's original and updated field values. Typically, therefore, you insert parameters with specially formatted names when you compose the SQL statements.

Using the Update SQL Editor

To create the SQL statements for an update component

1. Using the **Object Inspector**, select the name of the update object from the drop-down list for the dataset's *UpdateObject* property. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
2. Right-click the update object and select UpdateSQL Editor from the context menu. This displays the Update SQL editor. The editor creates SQL statements for the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you will want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Warning: Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

Understanding Parameter Substitution in Update SQL Statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string "OLD_", then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer's last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with "Smith" as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

Note: If you create SQL statements that contain parameters that do not refer the edited or original field values, the update object does not know how to bind their values. You can, however, do this manually, using the update object's Query property.

Composing Update SQL Statements

At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with "OLD_", the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some table types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

Using Multiple Update Objects

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its DataSet property to the name of the dataset.

Tip: When using multiple update objects, you can use *TBDEClientDataSet* instead of *TClientDataSet* with an external provider. This is because you do not need to set the source dataset's *UpdateObject* property.

The *DataSet* property for update objects is not available at design time in the **Object Inspector**. You can only set this property at runtime.

```
[Delphi]
UpdateSQL1.DataSet := Query1;
```

```
[C++]
UpdateSQL1->DataSet = Query1;
```

The update object uses this dataset to obtain original and updated field values for parameter substitution and, if it is a BDE-enabled dataset, to identify the session and database to use when applying the updates. So that parameter substitution will work correctly, the update object's *DataSet* property must be the dataset that contains the updated field values. When using the BDE-enabled dataset to cache updates, this is the BDE-enabled dataset itself. When using a client dataset, this is a client dataset that is provided as a parameter to the *BeforeUpdateRecord* event handler.

When the update object has not been assigned to the dataset's *UpdateObject* property, its SQL statements are not automatically executed when you call *ApplyUpdates*. To update records, you must manually call the update object from an OnUpdateRecord event handler (when using the BDE to cache updates) or a BeforeUpdateRecord event handler (when using a client dataset). In the event handler, the minimum actions you need to take are

- If you are using a client dataset to cache updates, you must be sure that the updates object's DatabaseName and SessionName properties are set to the DatabaseName and SessionName properties of the source dataset.
- The event handler must call the update object's ExecSQL or Apply method. This invokes the update object for each record that requires updating. For more information about executing update statements, see Executing the SQL statements.
- Set the event handler's *UpdateAction* parameter to *uaApplied* (*OnUpdateRecord*) or the *Applied* parameter to *True* (*BeforeUpdateRecord*).

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Warning: If you call an update object's *ExecSQL* or *Apply* method in an *OnUpdateRecord* event handler, be sure that you do not set the dataset's *UpdateObject* property to that update object. Otherwise, this will result in a second attempt to apply each record's update.

Executing the SQL Statements

When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code.

There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values:

- If the SQL statement to execute uses parameters, call the Apply method.
- If the SQL statement to execute does not use parameters, it is more efficient to call the ExecSQL method.

Note: If the SQL statement uses parameters other than the built-in types (for the original and updated field values), you must manually supply parameter values instead of relying on the parameter substitution provided by the *Apply* method. See Using an update component's Query property for information on manually providing parameter values.

Calling the Apply Method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Initial and edited field values for the record are bound to parameters in the appropriate SQL statement.

2 The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event or from a provider's *BeforeUpdateRecord* event handler.

Warning: If you use the dataset's *UpdateObject* property to associate dataset and update object, *Apply* is called automatically. In that case, do not call *Apply* in an *OnUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *Apply* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
[Delphi]
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DataSet := DeltaDS;
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    Apply(UpdateKind);
    Applied := True;
  end;
end;
```

```
[C++]
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS;
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->Apply(UpdateKind);
  Applied = true;
}
```

Executing an Update Statement

The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset).

Because *ExecSQL* does not bind parameter values, it is used primarily when the update object's SQL statements do not include parameters. You can use *Apply* instead, even when there are no parameters, but *ExecSQL* is more efficient because it does not check for parameters.

If the SQL statements include parameters, you can still call *ExecSQL,* but only after explicitly binding parameters. If you are using the BDE to cache updates, you can explicitly bind parameters by setting the update object's *DataSet* property and then calling its *SetParams* method. When using a client dataset to cache updates, you must

supply parameters to the underlying query object maintained by *TUpdateSQL*. For information on how to do this, see Using an update component's Query property.

Warning: If you use the dataset's *UpdateObject* property to associate dataset and update object, *ExecSQL* is called automatically. In that case, do not call *ExecSQL* in an *OnUpdateRecord* or *BeforeUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord and *BeforeUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *ExecSQL* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
[Delphi]
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ExecSQL(UpdateKind);
    Applied := True;
  end;
end;
```

```
[C++]
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->ExecSQL(UpdateKind);
  Applied = true;
}
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Using an Update Component's Query Property

The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL statements rather than relying on the update object's automatic parameter binding to old and new field values.

Note: The *Query* property is only accessible at runtime.

The *Query* property is indexed on a *TUpdateKind* value:

- Using an index of *ukModify* accesses the query that updates existing records.
- Using an index of *ukInsert* accesses the query that inserts new records.
- Using an index of *ukDelete* accesses the query that deletes records.

The following shows how to use the *Query* property to supply parameter values that can't be bound automatically:

```
[Delphi]
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
  with UpdateSQL1.Query[UpdateKind] do
  begin
    { Make sure the query has the correct DatabaseName and SessionName }
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ParamByName('TimeOfUpdate').Value = Now;
  end;
  UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
  Applied := True;
end;
```

```
[C++]
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS; // required for the automatic parameter substitution
  TQuery *pQuery = UpdateSQL1->Query[UpdateKind]; // access the query
  // make sure the query has the correct DatabaseName and SessionName
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  pQuery->DatabaseName = pSrcDS->DatabaseName;
  pQuery->SessionName = pSrcDS->SessionName;
  // now substitute values for custom parameters
  pQuery->ParamByName("TimeOfLastUpdate")->Value = Now();
  UpdateSQL1->Apply(UpdateKind); // now do automatic substitution and execute
  Applied = true;
}
```

Using TBatchMove

TBatchMove encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

The following topics describe how to work with a TBatchMove component:

- Creating a Batch Move Component
- Specifying a Batch Move Mode
- Mapping Data Types
- Executing a Batch Move
- Handling Batch Move Errors

Creating a Batch Move Component

To create a batch move component

1. Place a table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
2. Place the dataset to which to move records (called the *Destination* dataset) on the form or data module.
3. Place aTBatchMove component from the BDE category of the **Tool palette** in the data module or form, and set its *Name* property to a unique value appropriate to your application.
4. Set the Source property of the batch move component to the name of the table from which to copy, append, or update records. You can select tables from the drop-down list of available dataset components.
5. Set the Destination property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components.
   - If you are appending, updating, or deleting, *Destination* must represent an existing database table.
   - If you are copying a table and *Destination* represents an existing table, executing the batch move overwrites all of the current data in the destination table.
   - If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.
6. Set the Mode property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For information about these modes, see Specifying a batch move mode.
7. Optionally set the Transliterate property. If *Transliterate* is *True* (the default), character data is translated from the *Source* dataset's character set to the *Destination* dataset's character set as necessary.
8. Optionally set column mappings using the Mappings property. You need not set this property if you want batch move to match columns based on their position in the source and destination tables. For more information about mapping columns, see Mapping data types.
9. Optionally specify the ChangedTableName, KeyViolTableName, and ProblemTableName properties. Batch move stores problem records it encounters during the batch operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see Handling batch move errors.

Specifying a Batch Move Mode

The Mode property specifies the operation a batch move component performs:

Batch move modes

Property	Purpose
batAppend	Append records to the destination table.
batUpdate	Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table.

batCopy	Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated.
batDelete	Delete records in the destination table that match records in the source table.

Appending records

To append data, the destination dataset must represent an existing table. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

Updating records

To update data, the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

Appending and updating records

To append and update data the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise, data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset, if necessary.

Copying datasets

To copy a source dataset, the destination dataset should not represent an exist table. If it does, the batch move operation overwrites the existing table with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

Note: *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server as appropriate.

Deleting records

To delete data in the destination dataset, it must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

Mapping Data Types

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the Mappings property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name

in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a "best fit". It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of '5' to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see Handling batch move errors.

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset's server types. See the BDE online help file for the latest tables of mappings among server types.

Note: To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

Executing a Batch Move

Use the Execute method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
[Delphi]
BatchMoveAdd.Execute;
```

```
[C++]
BatchMoveAdd->Execute();
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing Execute from the context menu.

The MovedCount property keeps track of the number of records that are moved when a batch move executes.

The RecordCount property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

Handling Batch Move Errors

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. TBatchMove has a number of properties that report on and control error handling.

The AbortOnProblem property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The AbortOnKeyViol property indicates whether to abort the operation when a Paradox key violation occurs.

The ProblemCount property indicates the number of records that could not be handled in the destination table without a loss of data. If *AbortOnProblem* is *True*, this number is one, since the operation is aborted when an error occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- ChangedTableName, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- KeyViolTableName, if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.
- ProblemTableName, if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

Note: If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

The Data Dictionary

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the **Object Inspector** to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see Creating attribute sets for field components. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

Note: A programming interface to the Data Dictionary is available in the drintf unit (located in the lib directory). This interface supplies the following methods:

Data Dictionary interface

Routine	Use
DictionaryActive	Indicates if the data dictionary is active.
DictionaryDeactivate	Deactivates the data dictionary.

IsNullID	Indicates whether a given ID is a null ID
FindDatabaseID	Returns the ID for a database given its alias.
FindTableID	Returns the ID for a table in a specified database.
FindFieldID	Returns the ID for a field in a specified table.
FindAttrID	Returns the ID for a named attribute set.
GetAttrName	Returns the name an attribute set given its ID.
GetAttrNames	Executes a callback for each attribute set in the dictionary.
GetAttrID	Returns the ID of the attribute set for a specified field.
NewAttr	Creates a new attribute set from a field component.
UpdateAttr	Updates an attribute set to match the properties of a field.
CreateField	Creates a field component based on stored attributes.
UpdateField	Changes the properties of a field to match a specified attribute set.
AssociateAttr	Associates an attribute set with a given field ID.
UnassociateAttr	Removes an attribute set association for a field ID.
GetControlClass	Returns the control class for a specified attribute ID.
QualifyTableName	Returns a fully qualified table name (qualified by user name).
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name).
HasConstraints	Indicates whether the dataset has constraints in the dictionary.
UpdateConstraints	Updates the imported constraints of a dataset.
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary.

Tools for Working with the BDE

One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include:

- **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to
- Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases.
- Populate tables with data
- Create extended field attribute sets in the Data Dictionary or associate them with fields in your application.
- Create and manage BDE aliases.

SQL Explorer lets you do the following as well:

- Create SQL objects such as stored procedures on remote database servers.
- View the reconstructed text of SQL objects on remote database servers.
- Run SQL scripts.
- **SQL Monitor**: SQL Monitor lets you watch all of the communication that passes between the remote database server and the BDE. You can filter the messages you want to watch, limiting them to only the categories of interest. SQL Monitor is most useful when debugging your application.

- **Database Desktop**: If you are using Paradox or dBASE tables, Database Desktop lets you view and edit their data, create new tables, and restructure existing tables. Using Database Desktop affords you more control than using the methods of a *TTable* component (for example, it allows you to specify validity checks and language drivers). It provides the only mechanism for restructuring Paradox and dBASE tables other than making direct calls the BDE's API.

Working with ADO components

Working with ADO Components

The *dbGo* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The *dbGo* components encapsulate these ADO objects in the Delphi database architecture.

The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are wrapped by the *TADOConnection*, *TADOCommand*, and ADO dataset components. The ADO framework includes other "helper" objects, like the Field and Properties objects, but these are typically not used directly in dbGo applications and are not wrapped by dedicated components.

Before reading about the features peculiar to the *dbGo* components, you should familiarize yourself with the common features of database connection components and datasets.

The following topics describe the unique features of dbGo components and how to work with them:

- Overview of ADO components
- Connecting to ADO data stores
- Using TADODataSet
- Using Command Objects

Overview of ADO Components

The ADO page of the **Tool palette** hosts the *dbGo* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most *dbGo* components have direct counterparts in the components available for other data access mechanisms: a database connection component (*TADOConnection*) and various types of datasets. In addition, *dbGo* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

ADO components

Component	Use
TADOConnection	A database connection component that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TADODataSet	The primary dataset for retrieving and operating on data; *TADODataSet* can retrieve data from a single or multiple tables; can connect directly to a data store or use a *TADOConnection* component.
TADOTable	A table-type dataset for retrieving and operating on a recordset produced by a single database table; *TADOTable* can connect directly to a data store or use a *TADOConnection* component.
TADOQuery	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement; *TADOQuery* can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a *TADOConnection* component
TADOStoredProc	A stored procedure-type dataset for executing stored procedures; *TADOStoredProc* executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a *TADOConnection* component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets); *TADOCommand* can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a *TADOConnection* component.

Connecting to ADO Data Stores

dbGo applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party. If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect your application with the data store, use an ADO connection component (TADOConnection). Configure the ADO connection component to use one of the available ADO providers. Although *TADOConnection* is not strictly required, because ADO command and dataset components can establish connections directly using their *ConnectionString* property, you can use *TADOConnection* to share a single connection among several ADO components. This can reduce resource consumption, and allows you to create transactions that span multiple datasets.

Like other database connection components, *TADOConnection* provides support for

- Controlling connections
- Controlling server login
- Managing transactions
- Working with associated datasets
- Sending commands to the server
- Obtaining metadata

In addition to these features that are common to all database connection components, *TADOConnection* provides its own support for

- A wide range of options you can use to fine-tune the connection.
- The ability to list the command objects that use the connection.
- Additional events when performing common tasks.

Connecting to a Data Store Using TADOConnection

One or more ADO dataset and command components can share a single connection to a data store by using TADOConnection. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the **Object Inspector**. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataSet* component with a *TADOConnection* component.

```
[Delphi]
ADODataSet1.Connection := ADOConnection1;
```

```
[C++]
ADODataSet1->Connection = ADOConnection1;
```

The connection component represents an ADO connection object. Before you can use the connection object to establish a connection, you must identify the data store to which you want to connect. Typically, you provide information using the ConnectionString property. *ConnectionString* is a semicolon delimited string that lists one or more named connection parameters. These parameters identify the data store by specifying either the name of a file that contains the connection information or the name of an ADO provider and a reference identifying the data store. Use the following, predefined parameter names to supply this information:

Connection parameters

Parameter	Description
Provider	The name of a local ADO provider to use for the connection.
Data Source	The name of the data store.
File name	The name of a file containing connection information.
Remote Provider	The name of an ADO provider that resides on a remote machine.
Remote Server	The name of the remote server when using a remote provider.

Thus, a typical value of *ConnectionString* has the form

```
Provider=MSDASQL.1;Data Source=MQIS
```

Note: The connection parameters in *ConnectionString* do not need to include the *Provider* or *Remote Provider* parameter if you specify an ADO provider using the Provider property. Similarly, you do not need to specify the *Data Source* parameter if you use the DefaultDatabase property.

In addition, to the parameters listed above, *ConnectionString* can include any connection parameters peculiar to the specific ADO provider you are using. These additional connection parameters can include user ID and password if you want to hardcode the login information.

At design-time, you can use the Connection String Editor to build a connection string by selecting connection elements (like the provider and server) from lists. Click the ellipsis button for the *ConnectionString* property in the **Object Inspector** to launch the Connection String Editor, which is an ActiveX property editor supplied by ADO.

Once you have specified the *ConnectionString* property (and, optionally, the *Provider* property), you can use the ADO connection component to connect to or disconnect from the ADO data store, although you may first want to

use other properties to fine-tune the connection. When connecting to or disconnecting from the data store, *TADOConnection* lets you respond to a few additional events beyond those common to all database connection components..

Note: If you do not explicitly activate the connection by setting the connection component's *Connected* property to *True*, it automatically establishes the connection when the first dataset component is opened or the first time you use an ADO command component to execute a command.

Accessing the Connection Object

Use the ConnectionObject property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object.

Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

Fine-tuning a Connection

One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection.

The following topics describe the properties you can use to fine-tune the connection:

- Forcing asynchronous connections
- Controlling time-outs
- Indicating the types of operations the connection supports
- Specifying whether the connection automatically initiates transactions

Forcing Asynchronous Connections

Use the ConnectOptions property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened.

By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

The example routines below enable and disable asynchronous connections in the specified connection component:

```
[Delphi]
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;
procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
```

```
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

```
[C++]
void __fastcall TForm1::AsyncConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coAsyncConnect;
  ADOConnection1->Open();
}
void __fastcall TForm1::ServerChoiceConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coConnectUnspecified;
  ADOConnection1->Open();
}
```

Controlling Timeouts

You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the ConnectionTimeout and CommandTimeout properties.

ConnectionTimeout specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in ConnectionTimeout, the connection attempt is canceled:

```
[Delphi]
with ADOConnection1 do begin
  ConnectionTimeout := 10 {seconds};
  Open;
end;
```

```
[C++]
  ADOConnection1->ConnectionTimeout = 10; // seconds
  ADOConnection1->Open();
```

CommandTimeout specifies the amount of time, in seconds, before an attempted command times out. If a command initiated by a call to the Execute method does not successfully complete prior to expiration of the time specified in CommandTimeout, the command is canceled and ADO generates an exception:

```
[Delphi]
with ADOConnection1 do begin
  CommandTimeout := 10 {seconds};
  Execute("DROP TABLE Employee1997", cmdText, []);
end;
```

```
[C++]
  ADOConnection1->ConnectionTimeout = 10;
  ADOConnection1->Execute("DROP TABLE Employee1997", cmdText, TExecuteOptions());
```

Indicating the Types of Operations the Connection Supports

ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection.

Use the Mode property to indicate the connection mode. The possible values are listed in the following table:

ADO connection modes

Connect Mode	Meaning
cmUnknown	Permissions are not yet set for the connection or cannot be determined.
cmRead	Read-only permissions are available to the connection.
cmWrite	Write-only permissions are available to the connection.
cmReadWrite	Read/write permissions are available to the connection.
cmShareDenyRead	Prevents others from opening connections with read permissions.
cmShareDenyWrite	Prevents others from opening connection with write permissions.
cmShareExclusive	Prevents others from opening connection.
cmShareDenyNone	Prevents others from opening connection with any permissions.

The possible values for *Mode* correspond to the *ConnectModeEnum* values of the *Mode* property on the underlying ADO connection object. See the Microsoft Data Access SDK help for more information on these values.

Specifying Whether the Connection Automatically Initiates Transactions

Use the Attributes property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started.

Attributes is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts.

Check whether either retaining commits or retaining aborts is enabled using the in operator. Enable retaining commits or aborts by adding the appropriate value to the attributes property; disable them by subtracting the value. The example routines below respectively enable and disable retaining commits in an ADO connection component.

```
[Delphi]
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;
procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
```

```
  end;
end;
```

```
[C++]
void __fastcall TForm1::RetainingCommitsOnButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (!ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() << xaCommitRetaining;
  ADOConnection1->Open()
}
void __fastcall TForm1::RetainingCommitsOffButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() >> xaCommitRetaining;
  ADOConnection1->Open()
}
```

Accessing the Connection's Datasets

Like other database connection components, you can access the datasets associated with the connection using the *DataSets* and *DataSetCount* properties. However, *dbGo* also includes TADOCommand objects, which are not datasets, but which maintain a similar relationship to the connection component.

You can use the Commands and CommandCount properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component.

Commands is a zero-based array of references to ADO command components. *CommandCount* provides a total count of all of the commands listed in *Commands*. You can use these properties together to iterate through all the commands that use a connection component, as illustrated in the following code:

```
[Delphi]
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;
```

```
[C++]
for (int i = 0; i < ADOConnection2->CommandCount; i++)
  ADOConnection2->Commands[i]->Execute();
```

ADO Connection Events

In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.

Events when establishing a connection

In addition to the *BeforeConnect* and *AfterConnect* events that are common to all database connection components, *TADOConnection* also generates an *OnWillConnect* and *OnConnectComplete* event when establishing a connection. These events occur after the *BeforeConnect* event.

- OnWillConnect occurs before the ADO provider establishes a connection. It lets you make last minute changes to the connection string, provide a user name and password if you are handling your own login support, force an asynchronous connection, or even cancel the connection before it is opened.
- OnConnectComplete occurs after the connection is opened. Because *TADOConnection* can represent asynchronous connections, you should use *OnConnectComplete*, which occurs after the connection is opened or has failed due to an error condition, instead of the *AfterConnect* event, which occurs after the connection component instructs the ADO provider to open a connection, but not necessarily after the connection is opened.

Events when disconnecting

In addition to the *BeforeDisconnect* and *AfterDisconnect* events common to all database connection components, *TADOConnection* also generates an OnDisconnect event after closing a connection. *OnDisconnect* occurs after the connection is closed but before any associated datasets are closed and before the *AfterDisconnect* event.

Events when managing transactions

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method has been successfully completed at the data store.

- The OnBeginTransComplete event occurs when the data store has successfully started a transaction after a call to the BeginTrans method.
- The OnCommitTransComplete event occurs after a transaction is successfully committed due to a call to CommitTrans.
- The OnRollbackTransComplete event occurs after a transaction is successfully aborted due to a call to *RollbackTrans*.

Other events

ADO connection components introduce two additional events you can use to respond to notifications from the underlying ADO connection object:

- The OnExecuteComplete event occurs after the connection component executes a command on the data store (for example, after calling the *Execute* method). *OnExecuteComplete* indicates whether the execution was successful.
- The OnInfoMessage event occurs when the underlying connection object provides detailed information after an operation is completed. The *OnInfoMessage* event handler receives the interface to an ADO Error object that contains the detailed information and a status code indicating whether the operation was successful.

Using ADO datasets

ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Understanding Datasets, using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features.

In addition to the common dataset features, all ADO datasets add properties, events, and methods for the following:

- Connecting to an ADO datastore
- Accessing the underlying Recordset object
- Filtering records based on bookmarks
- Fetching records asynchronously
- Performing batch updates (caching updates)
- Using files on disk to store data

There are four ADO datasets:

- *TADOTable*, a table-type dataset that represents all of the rows and columns of a single database table.
- *TADOQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any.
- *TADOStoredProc*, a stored procedure-type dataset that executes a stored procedure defined on a database server.
- *TADODataSet*, a general-purpose dataset that includes the capabilities of the other three types. See Using TADODataSet for a description of features unique to *TADODataSet*.

Note: When using ADO to access database information, you do not need to use a dataset such as *TADOQuery* to represent SQL commands that do not return a cursor. Instead, you can use *TADOCommand*, a simple component that is not a dataset. For details on *TADOCommand*, see Using Command Objects.

Connecting an ADO Dataset to a Data Store

ADO datasets can connect to an ADO data store either collectively or individually.

When connecting datasets collectively, set the Connection property of each dataset to a TADOConnection component. Each dataset then uses the ADO connection component's connection.

```
[Delphi]
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
```

```
[C++]
ADODataSet1->Connection = ADOConnection1;
ADODataSet2->Connection = ADOConnection1;
...
```

Among the advantages of connecting datasets collectively are:

- The datasets share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The datasets can participate in transactions.

For more information on using *TADOConnection* see Connecting to ADO data stores.

When connecting datasets individually, set the ConnectionString property of each dataset. Each dataset that uses *ConnectionString* establishes its own connection to the data store, independent of any other dataset connection in the application.

The *ConnectionString* property of ADO datasets works the same way as the *ConnectionString* property of *TADOConnection*: it is a set of semicolon-delimited connection parameters such as the following:

```
[Delphi]
ADODataSet1.ConnectionString := "Provider=YourProvider;Password=SecretWord;" +
  "User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;" +
  "Initial Catalog=Employee";
```

```
[C++]
ADODataSet1->ConnectionString = "Provider=YourProvider;Password=SecretWord;";
ADODataSet1->ConnectionString += "User ID=JaneDoe;SERVER=PURGATORY";
ADODataSet1->ConnectionString += "UID=JaneDoe;PWD=SecretWord;"
ADODataSet1->ConnectionString += "Initial Catalog=Employee";
```

At design time you can use the Connection String Editor to help you build the connection string. For more information about connection strings, see Connecting to a data store using TADOConnection.

Working with Record Sets

The Recordset property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

The RecordSetState property indicates the current state of the underlying recordset object. *RecordsetState* corresponds to the *State* property of the ADO recordset object. The value of *RecordsetState* is either *stOpen*, *stExecuting*, or *stFetching*. (*TObjectState*, the type of the *RecordsetState* property, defines other values, but only *stOpen*, *stExecuting,* and *stFetching* pertain to recordsets.) A value of *stOpen* indicates that the recordset is currently idle. A value of stExecuting indicates that it is executing a command. A value of stFetching indicates that it is fetching rows from the associated table (or tables).

Use *RecordsetState* values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the RecordsetState property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

Filtering Records Based On Bookmarks

ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records. Also like other datasets, ADO datasets let you use filters to limit the available records in the dataset. ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.

To filter on a set of bookmarks

1 Use the Bookmark method to mark the records you want to include in the filtered dataset.

2 Call the FilterOnBookmarks method to filter the dataset so that only the bookmarked records appear.

This process is illustrated below:

```
[Delphi]
procedure TForm1.Button1Click(Sender: TObject);
var
BM1, BM2: TBookmarkStr;
begin
with ADODataSet1 do begin
BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
MoveBy(3);
BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
end;
end;
```

```
[C++]
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TBookmarkStr BM1;
TBookmarkStr BM2;
BM1 = ADODataSet1->Bookmark;
  BMList->Add(BM1);
ADODataSet1->MoveBy(3);
BM2 = ADODataSet1->Bookmark;
  BMList->Add(BM2);
  ADODataSet1->FilterOnBookmarks(ARRAYOFCONST((BM1,BM2)));
}
```

Note that the example above also adds the bookmarks to a list object named BMList. This is necessary so that the application can later free the bookmarks when they are no longer needed.

Fetching Records Asynchronously

Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store.

To control whether the dataset fetches data asynchronously, if it fetches data at all, use the ExecuteOptions property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, ExecuteOptions governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*.

ExecuteOptions is a set that includes zero or more of the following values:

Execution options for ADO datasets

Execute Option	Meaning
eoAsyncExecute	The command or data fetch operation is executed asynchronously.
eoAsyncFetch	The dataset first fetches the number of records specified by the *CacheSize* property synchronously, then fetches any remaining rows asynchronously.
eoAsyncFetchNonBlocking	Asynchronous data fetches or command execution do not block the current thread of execution.

eoExecuteNoRecords	A command or stored procedure that does not return data. If any rows are retrieved, they are discarded and not returned.

Using Batch Updates

One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in Using a client dataset to cache updates.

However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features:

Comparison of ADO and client dataset cached updates

ADO dataset	TClientDataSet	Description
LockType	Not used: client datasets always cache updates	Specifies whether the dataset is opened in batch update mode.
CursorType	Not used: client datasets always work with an in-memory snapshot of data	Specifies how isolated the ADO dataset is from changes on the server.
RecordStatus	UpdateStatus	Indicates what update, if any, has occurred on the current row. *RecordStatus* provides more information than *UpdateStatus*.
FilterGroup	StatusFilter	Specifies which type of records are available. *FilterGroup* provides a wider variety of information.
UpdateBatch	ApplyUpdates	Applies the cached updates back to the database server. Unlike *ApplyUpdates*, *UpdateBatch* lets you limit the types of updates to be applied.
CancelBatch	CancelUpdates	Discards pending updates, reverting to the original values. Unlike *CancelUpdates*, *CancelBatch* lets you limit the types of updates to be canceled.

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates

Opening the Dataset in Batch Update Mode

To open an ADO dataset in batch update mode, it must meet these criteria:

1. The component's CursorType property must be *ctKeySet* (the default property value) or *ctStatic*.
2. The LockType property must be *ltBatchOptimistic*.
3. The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's CommandText property (for *TADODataSet*) or the SQL property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that

returns a result set. These properties can be set at design-time through the **Object Inspector** or programmatically at runtime. The example below shows the preparation of a *TADODataSet* component for batch update mode.

```
[Delphi]
with ADODataSet1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

```
[C++]
ADODataSet1->CursorLocation = clUseClient;
ADODataSet1->CursorType = ctStatic;
ADODataSet1->LockType = ltBatchOptimistic;
ADODataSet1->CommandType = cmdText;
ADODataSet1->CommandText = "SELECT * FROM Employee";
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

Inspecting the Update Status of Individual Rows

Determine the update status of a given row by making it current and then inspecting the RecordStatus property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
[Delphi]
if (rsNew in ADOQuery1.RecordStatus) then
begin
...
end;
else
if (rsDeleted in ADOQuery1.RecordStatus) then
begin
...
else
```

```
[C++]
switch (ADOQuery->RecordStatus)
{
  case rsUnmodified:
    StatusBar1->Panels->Items[0]->Text = "Unchanged record";
    break;
  case rsModified:
    StatusBar1->Panels->Items[0]->Text = "Changed record";
    break;
  case rsDeleted:
    StatusBar1->Panels->Items[0]->Text = "Deleted record";
    break;
  case rsNew:
    StatusBar1->Panels->Items[0]->Text = "New record";
    break;
}
```

Filtering Multiple Rows Based On Update Status

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the FilterGroup property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
[Delphi]
FilterGroup := fgPendingRecords;
Filtered := True;
```

```
[C++]
FilterGroup = fgPendingRecords;
Filtered = true;
```

Note: For the *FilterGroup* property to have an effect, the ADO dataset component's Filtered property must be set to *True*.

Applying the Batch Updates to Base Tables

Apply pending data changes that have not yet been applied or canceled by calling the UpdateBatch method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *arAll* is passed, only a subset of the pending changes are applied. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
[Delphi]
ADODataSet1.UpdateBatch(arCurrent);
```

```
[C++]
ADODataSet1->UpdateBatch(arCurrent);
```

Canceling Batch Updates

Cancel pending data changes that have not yet been canceled or applied by calling the CancelBatch method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
[Delphi]
ADODataSet1.CancelBatch;
```

```
[C++]
ADODataSet1->CancelBatch(arAll);
```

Loading Data from and Saving Data to Files

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported.

Save the data to a file using the SaveToFile method. *SaveToFile* takes two parameters, the name of the file to which data is saved, and, optionally, the format (ADTG or XML) in which to save the data. Indicate the format for the saved file by setting the Format parameter to *pfADTG* or *pfXML*. If the file specified by the *FileName* parameter already exists, *SaveToFile* raises an *EOleException*.

Retrieve the data from file using the LoadFromFile method. *LoadFromFile* takes a single parameter, the name of the file to load. If the specified file does not exist, *LoadFromFile* raises an *EOleException* exception. On calling the *LoadFromFile* method, the dataset component is automatically activated.

In the example below, the first procedure saves the dataset retrieved by the *TADODataSet* component *ADODataSet1* to a file. The target file is an ADTG file named SaveFile, saved to a local drive. The second procedure loads this saved file into the *TADODataSet* component *ADODataSet2*.

```
[Delphi]
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists("c:\SaveFile")) then
  begin
    DeleteFile("c:\SaveFile");
    StatusBar1.Panels[0].Text := "Save file deleted!";
  end;
  ADODataSet1.SaveToFile("c:\SaveFile", pfADTG);
end;
procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists("c:\SaveFile")) then
    ADODataSet2.LoadFromFile("c:\SaveFile")
  else
    StatusBar1.Panels[0].Text := "Save file does not exist!";
end;
```

```
[C++]
void __fastcall TForm1::SaveBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
  {
    DeleteFile("c:\\SaveFile");
    Statusbar1->Panels->Items[0]->Text = "Save file deleted!";
  }
  ADODataSet1->SaveToFile("c:\\SaveFile");
}
void __fastcall TForm1::LoadBtnClick(TObject *Sender)
```

```
{
  if (FileExists("c:\\SaveFile"))
    ADODataSet1->LoadFromFile("c:\\SaveFile");
  else
    Statusbar1->Panels->Items[0]->Text = "Save file does not exist!";
}
```

The datasets that save and load the data need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

Using TADODataSet

TADODataSet is a general-purpose dataset for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataSet* is not a table-type, query-type, or stored procedure-type dataset. Instead, it can function as any of these types:

- Like a table-type dataset, *TADODataSet* lets you represent all of the rows and columns of a single database table. To use it in this way, set the CommandType property to *cmdTable* and the CommandText property to the name of the table. *TADODataSet* supports table-type tasks such as
- Assigning indexes to sort records or form the basis of record-based searches. In addition to the standard index properties and methods, *TADODataSet* lets you sort using temporary indexes by setting the Sort property. Indexed-based searches performed using the Seek method use the current index.
- Emptying the dataset. The DeleteRecordsDeleteRecords method provides greater control than related methods in other table-type datasets, because it lets you specify what records to delete.

The table-type tasks supported by *TADODataSet* are available even when you are not using a *CommandType* of *cmdTable*.

- Like a query-type dataset, *TADODataSet* lets you specify a single SQL command that is executed when you open the dataset. To use it in this way, set the CommandType property to *cmdText* and the CommandText property to the SQL command you want to execute. At design time, you can double-click on the *CommandText* property in the **Object Inspector** to use the Command Text editor for help in constructing the SQL command. *TADODataSet* supports query-type tasks such as
- Using parameters in the query text.
- Setting up master/detail relationships using parameters.
- Preparing the query in advance to improve performance by setting the Prepared property to *True*.
- Like a stored procedure-type dataset, *TADODataSet* lets you specify a stored procedure that is executed when you open the dataset. To use it in this way, set the CommandType property to *cmdStoredProc* and the CommandText property to the name of the stored procedure. *TADODataSet* supports stored procedure-type tasks such as
- Working with stored procedure parameters.
- Fetching multiple result sets.
- Preparing the stored procedure in advance to improve performance by setting the Prepared property to *True*.

In addition, *TADODataSet* lets you work with data stored in files by setting the *CommandType* property to *cmdFile* and the *CommandText* property to the file name.

Before you set the *CommandText* and *CommandType* properties, you should link the *TADODataSet* to a data store by setting the Connection or ConnectionString property. This process is described in Connecting an ADO dataset to a data store. As an alternative, you can use an RDS DataSpace object to connect the *TADODataSet* to an ADO-based application server. To use an RDS DataSpace object, set the RDSConnection property to a TRDSConnection object.

Using Command Objects

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the TADOCommand component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its *Execute* method, however, it is capable of returning a result set that can be assigned to the *RecordSet* property of an ADO dataset component.

In general, working with *TADOCommand* is very similar to working with *TADODataSet*, except that you can't use the standard dataset methods to fetch data, navigate records, edit data, and so on. *TADOCommand* objects connect to a data store in the same way as ADO datasets. See Connecting an ADO dataset to a data store for details.

The following topics provide details on how to specify and execute commands using *TADOCommand*:

- Specifying the command
- Using Command objects
- Canceling commands
- Retrieving result sets with commands
- Handling command parameters

Specifying the Command

Specify commands for a *TADOCommand* component using the CommandText property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the CommandType property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the **Object Inspector**. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
[Delphi]
with ADOCommand1 do begin
  CommandText := "AddEmployee";
  CommandType := cmdStoredProc;
...
end;
```

```
[C++]
ADOCommand1->CommandText = "AddEmployee";
ADOCommand1->CommandType = cmdStoredProc;
...
```

If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*.

CommandText can contain the text of an SQL query that includes parameters or the name of a stored procedure that uses parameters. You must then supply parameter values, which are bound to the parameters before executing the command. See Handling command parameters for details.

Using the Execute Method

Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See Connecting an ADO dataset to a data store for details.

To execute the command, call the Execute method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command.

For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:

```
[Delphi]
with ADOCommand1 do begin
  CommandText := "UpdateInventory";
  CommandType := cmdStoredProc;
  Execute;
end;
```

```
[C++]
ADOCommand1->CommandText = "UpdateInventory";
ADOCommand1->CommandType = cmdStoredProc;
ADOCommand1->Execute();
```

Other versions of *Execute* let you provide parameter values using a Variant array, and to obtain the number of records affected by the command.

For information on executing commands that return a result set, see Retrieving result sets with commands.

Canceling Commands

If you are executing the command asynchronously, then after calling Execute you can abort the execution by calling the Cancel method:

```
[Delphi]
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;
procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

```
[C++]
void __fastcall TDataForm::ExecuteButtonClick(TObject *Sender)
{
  ADOCommand1->Execute();
}
void __fastcall TDataForm::CancelButtonClick(TObject *Sender)
{
  ADOCommand1->Cancel();
}
```

The *Cancel* method only has an effect if there is a command pending and it was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called but the command has not yet been completed or timed out.

A command times out if it is not completed or canceled before the number of seconds specified in the CommandTimeout property expire. By default, commands time out after 30 seconds.

Retrieving Result Sets with Commands

Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the Execute command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the ADO _RecordSet interface.

The most convenient way to work with this interface is to assign it to the RecordSet property of an ADO dataset.

For example, the following code uses *TADOCommand* (*ADOCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the *RecordSet* property of a *TADODataSet* component (*ADODataSet1*).

```
[Delphi]
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

```
[C++]
ADOCommand1->CommandText = "SELECT Company, State ";
ADOCommand1->CommandText += "FROM customer ";
ADOCommand1->CommandText += "WHERE State = :StateParam";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("StateParam")->Value = "HI";
ADOCommand1->Recordset = ADOCommand1->Execute();
```

As soon as the result set is assigned to the ADO dataset's *Recordset* property, the dataset is automatically activated and the data is available.

Handling Command Parameters

There are two ways in which a *TADOCommand* object may use parameters:

- The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCommand* works like using a parameterized query in an ADO dataset.
- The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters work much the same using *TADOCommand* as with an ADO dataset.

There are two ways to supply parameter values when working with *TADOCommand*: you can supply them when you call the Execute method, or you can specify them ahead of time using the Parameters property.

The *Execute* method is overloaded to include versions that take a set of parameter values as a Variant array. This is useful when you want to supply parameter values quickly without the overhead of setting up the *Parameters* property:

```
[Delphi]
ADOCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

```
[C++]
Variant Values[2];
Values[0] = Edit1->Text;
Values[1] = Date();
ADOCommand1.Execute(VarArrayOf(Values,1));
```

When working with stored procedures that return output parameters, you must use the *Parameters* property instead. Even if you do not need to read output parameters, you may prefer to use the *Parameters* property, which lets you supply parameters at design time and lets you work with *TADOCommand* properties in the same way you work with the parameters on datasets.

When you set the *CommandText* property, the *Parameters* property is automatically updated to reflect the parameters in the query or those used by the stored procedure. At design-time, you can use the Parameter Editor to access parameters, by clicking the ellipsis button for the *Parameters* property in the **Object Inspector**. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.

```
[Delphi]
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName("NewValueParam").Value := 57;
  Execute
end;
```

```
[C++]
ADOCommand1->CommandText = "INSERT INTO Talley ";
ADOCommand1->CommandText += "(Counter) ";
ADOCommand1->CommandText += "VALUES (:NewValueParam)";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("NewValueParam")->Value = 57;
ADOCommand1->Execute()
```

Using unidirectional datasets

Using Unidirectional Datasets

dbExpress is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by TDataSet are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the First and Next methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The CanModify property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use unidirectional datasets to update data using an SQL UPDATE command or provide conventional editing support by using a dbExpress-enabled client dataset or connecting the dataset to a client dataset .
- There is no support for filters, because filters work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.
- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are the fastest data access mechanism, and very simple to use and deploy.

The following topics describe unidirectional datasets in greater detail:

- Types of unidirectional datasets
- Connecting to the database server
- Specifying what data to display
- Fetching the data
- Executing commands that do not return records

- Setting up master/detail linked cursors
- Accessing Schema Information
- Debugging dbExpress applications

Types of Unidirectional Datasets

The *dbExpress* category of the **Tool palette** contains four types of unidirectional dataset: TSQLDataSet, TSQLQuery, TSQLTable, and TSQLStoredProc.

TSQLDataSet is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

TSQLQuery is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any.

TSQLTable is a table-type dataset that represents all of the rows and columns of a single database table.

TSQLStoredProc is a stored procedure-type dataset that executes a stored procedure defined on a database server.

Note: The *dbExpress* page also includes TSimpleDataSet, which is not a unidirectional dataset. Rather, it is a client dataset that uses a unidirectional dataset internally to access its data.

Connecting to the Database Server

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server.

The connection to a database server is represented by a separate TSQLConnection component. You work with *TSQLConnection* like any other database connection component.

To use *TSQLConnection* to connect a unidirectional dataset to a database server, set the SQLConnection property. At design time, you can choose the SQL connection component from a drop-down list in the **Object Inspector**. If you make this assignment at runtime, be sure that the connection is active:

```
[Delphi]
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

```
[C++]
SQLDataSet1->SQLConnection = SQLConnection1;
SQLConnection1->Connected = true;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers. However, you may want to use a separate connection for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the MaxStmtsPerConn property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the AutoClone property to *False*.

Before you assign the *SQLConnection* property, you will need to set up the TSQLConnection component so that it identifies the database server and any required connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on).

Setting Up TSQLConnection

In order to describe a database connection in sufficient detail for TSQLConnection to open a connection, you must identify both the driver to use and a set of connection parameters the are passed to that driver.

Identifying the driver

The driver is identified by the DriverName property, which is the name of an installed *dbExpress* driver, such as INTERBASE, INFORMIX, ORACLE, MYSQL, MSSQL, or DB2. The driver name is associated with two files:

- The *dbExpress* driver. This can be either a dynamic-link library with a name like dbexpint.dll, dbexpora.dll, dbexpmysql.dll, or dbexpdb2.dll, or a compiled unit that you can statically link into your application (dbexpint.dcu, dbexpora.dcu, dbexpmys.dcu, or dbexpdb2.dcu).
- The dynamic-link library provided by the database vendor for client-side support.

The relationship between these two files and the database name is stored in a file called dbxdrivers.ini, which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers.ini when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the LibraryName and VendorLib properties to the names of the associated dlls. Once *LibraryName* and *VendorLib* have been set, your application does not need to rely on dbxdrivers.ini. (That is, you do not need to deploy dbxdrivers.ini with your application unless you set the *DriverName* property at runtime.)

Specifying connection parameters

The Params property is a string list that lists name/value pairs. Each pair has the form *Name*=*Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with ORACLE it is the entry in TNSNames.ora, while with DB2, it is the client-side node name.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the **Object Inspector** to edit the parameters using the String List editor. At runtime, use the Params.Values property to assign values to individual parameters.

Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. You can name *dbExpress* database and parameter combinations, which are then saved in a file called dbxconnections.ini. The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the ConnectionName property to a valid connection name. Setting *ConnectionName* automatically sets the

DriverName and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections.ini file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections.ini. Then, when you deploy your application, it loads these values from a separate version of dbxconnections.ini that uses the "real" database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the LoadParamsOnConnect property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini when the connection is opened.
- Call the LoadParamsFromIniFile method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections.ini file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections.ini:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to dbxconnections.ini.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.ini.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.ini.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

Specifying What Data to Display

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a TSQLDataSet component, use the CommandType property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.

- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

The following topics describe how you can specify a set of records for each type of source:

- Representing the results of a query
- Representing the records in a table
- Representing the results of a stored procedure

Note: You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see Fetching metadata into a unidirectional dataset.

Representing the Results of a Query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

When using *TSQLDataSet*, set the CommandType property to *ctQuery* and assign the text of the query statement to the CommandText property. When using *TSQLQuery*, assign the query to the SQL property instead. These properties work the same way for all general-purpose or query-type datasets. Specifying the query discusses them in greater detail.

When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. Using parameters in queries and supplying values for those parameters is discussed in Using parameters in queries.

SQL defines queries such as UPDATE queries that perform actions on the server but do not return records. Such queries are discussed in Executing commands that do not return records.

Representing the Records in a Table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

Note: If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the CommandType property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- CommandText specifies the name of the database table that the *TSQLDataSet* object should represent.
- SortFieldNames lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
[Delphi]
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

```
[C++]
SQLDataSet1->CommandType = ctTable;
SQLDataSet1->CommandText = "Employee";
SQLDataSet1->SortFieldNames = "HireDate,Salary"
```

TSQLDataSet generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the TableName property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the IndexName property to the name of an index defined on the server that imposes the order you want.
- Set the IndexFieldNames property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

Representing the Results of a Stored Procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

When using TSQLDataSet, to specify a stored procedure:

- Set the CommandType property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the CommandText property:

```
[Delphi]
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

```
[C++]
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "MyStoredProcName";
```

When using TSQLStoredProc, you need only specify the name of the stored procedure as the value of the StoredProcName property.

```
[Delphi]
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

```
[C++]
SQLStoredProc1->StoredProcName = "MyStoredProcName";
```

After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure. See Working with stored procedure parameters for information about working with stored procedure parameters.

Fetching the Data

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

One way is to set the Activeproperty to *True*, either at design time in the **Object Inspector**, or in code at runtime:

```
[Delphi]
CustQuery.Active := True;
```

```
[C++]
CustQuery->Active = true;
```

Another way is to call the Open method at runtime,

```
[Delphi]
CustQuery.Open;
```

```
[C++]
CustQuery->Open();
```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be "prepared". Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its Prepared property to *True*.

```
[Delphi]
CustQuery.Prepared := True;
```

```
[C++]
CustQuery->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the NextRecordSet method:

```
[Delphi]
var
  DataSet2: TCustomSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

```
[C++]
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

NextRecordSet returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

Executing Commands That Do Not Return Records

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

Note: If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See Sending commands to the server for details.

The following topics discuss how to create and execute a command that does not return any records:

- Specifying the command to execute
- Executing the command

In addition, the topic,Creating and modifying server metadata, discusses some of the SQL commands that do not return datasets:

Specifying the Command to Execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using TSQLDataSet, use the CommandType and CommandText properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.
- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using TSQLQuery, use the SQL property to specify the SQL statement to pass to the server.

When using TSQLStoredProc, use the StoredProcName property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records.

Executing the Command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

The ExecSQL method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
[Delphi]
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

```
[C++]
FixTicket->CommandText = "DELETE FROM TrafficViolations WHERE (TicketID = 1099)";
FixTicket->ExecSQL();
```

The ExecProc method if the dataset is an instance of *TSQLStoredProc*.

```
[Delphi]
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

```
[C++]
SQLStoredProc1->StoredProcName = "MyCommandWithNoResults";
SQLStoredProc1->ExecProc();
```

```
[Delphi]
```

If you are executing the query or stored procedure multiple times, it is a good idea to set the Prepared property to *True*.

Creating and Modifying Server Metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing. In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset such as *TClientDataSet*.

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

```
[Delphi]
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
       END';
  ExecSQL;
end;
```

```
[C++]
SQLDataSet1->ParamCheck = false;
SQLDataSet1->CommandType = ctQuery;
SQLDataSet1->CommandText = "CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) RETURNS
```

```
(PROJ_ID CHAR(5)) AS BEGIN FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT WHERE EMP_NO = :EMP_NO
INTO :PROJ_ID DO SUSPEND; END";
SQLDataSet1->ExecSQL();
```

Setting Up Master/detail Linked Cursors

There are two ways to use linked cursors to set up a master/detail relationship with a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the "many" in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the "one" in the one-to-many relationship).

TSQLDataSet and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all query-type datasets. For details on creating master/detail relationships with query-type datasets, see Establishing master/detail relationships using parameters.

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the MasterSource and MasterFields properties, just as you would with any other table-type dataset. For details on creating master/detail relationships with table-type datasets, see Creating Master/detail Relationships.

Accessing Schema Information

There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

The simplest way to obtain this metadata is to use the methods of TSQLConnection. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This technique is the same as the way you fill lists with metadata for any other database connection component. These methods are described in Obtaining metadata.

If you require more detailed schema information, you can populate a unidirectional dataset with metadata. Instead of a simple list, the unidirectional dataset is filled with schema information, where each record represents a single table, stored procedure, index, field, or parameter. See Fetching metadata into a unidirectional dataset for details on populating a unidirectional dataset with schema information.

Fetching Metadata into a Unidirectional Dataset

To populate a unidirectional datasets with metadata from the database server, you must first indicate what data you want to see, using the SetSchemaInfo method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see The structure of metadata datasets.
- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as 'Cust%', which uses the wildcards '%' (to match a string of arbitrary characters of any length) and '_' (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil.

If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection's TableScope property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
[Delphi]
SQLDataSet1.SetSchemaInfo(stSysTable, "", "");
```

```
[C++]
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named 'MyProc'. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters ('inName', 'outValue' and so on). You could call *SetSchemaInfo* as follows:

```
[Delphi]
SQLDataSet1.SetSchemaInfo(stProcedureParams, "MyProc", "in%");
```

```
[C++]
SQLDataSet1->SetSchemaInfo(stProcedureParams, "MyProc", "in%");
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the CommandText property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

The Structure of Metadata Datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

Columns in tables of metadata listing tables

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.

CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

Columns in tables of metadata listing stored procedures

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

Columns in tables of metadata listing fields

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas.

COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on).
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values.
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank (0 means the field requires a value).

Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

Columns in tables of metadata listing indexes

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d).
FILTER	ftString	Describes a filter condition that limits the indexed records.

Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

Columns in tables of metadata listing parameters

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.

SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property.
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas.
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank (0 means the parameter requires a value).

Information about Oracle packages

Columns in tables of metadata listing stored procedures

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the package. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the package.
OBJECT_NAME	ftString	The name of the package. This field determines the sort order of the dataset.

Debugging dbExpress Applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

Using TSQLMonitor to monitor SQL commands

TSQLConnection uses a companion component, TSQLMonitor, to intercept these messages and save them in a string list. *TSQLMonitor* works much like the SQL monitor utility that you can use with the BDE, except that it monitors only those commands involving a single *TSQLConnection* component rather than all commands managed by *dbExpress*.

To use TSQLMonitor

1. Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
2. Set its SQLConnection property to the *TSQLConnection* component.
3. Set the SQL monitor's Active property to *True*.

Flags for monitoring SQL commands

Flag	Meaning
traceUNKNOWN	All SQL commands.
traceQPREPARE	prepared queries sent to the server.
traceQEXECUTE	Queries to be executed by the server. Note that a single statement may be prepared once and executed several times with different parameter bindings.
traceERROR	Error messages returned by the server. The error message may include an error code, depending on the server.
traceSTMT	Operations to be performed such as ALLOCATE, PREPARE, EXECUTE, and FETCH.
traceCONNECT	Operations associated with connecting and disconnecting to databases, including allocation of connection handles and freeing connection handles, if required by server.
traceTRANSACT	Transaction operations such as BEGIN, COMMIT, and ROLLBACK (ABORT).
traceBLOB	Operations on Binary Large Object (BLOB) data, including STORE BLOB, GET BLOB HANDLE, and so on.
traceMISC	commands not covered by any other flag.
traceVENDOR	API function calls to the server. For example, ORLON for Oracle, ISC_ATTACH for InterBase.
traceDATAIN	Parameter data sent to servers when doing INSERTs or UPDATEs.
traceDATAOUT	Data retrieved from servers.

As SQL commands are sent to the server, the SQL monitor's TraceList property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the FileName property and then setting the AutoSave property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the OnLogTrace event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

```
[Delphi]
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
```

```
  end;
end;
```

```
[C++]
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender, void *CBInfo)
{
  TSQLMonitor *pMonitor = dynamic_cast<TSQLMonitor *>(Sender);
  if (pMonitor->TraceCount == 10)
  {
    // build unique file name
    AnsiString LogFileName = "c:\\log";
    LogFileName = LogFileName + IntToStr(pMonitor->Tag);
    LogFileName = LogFileName + ".txt"
    pMonitor->Tag = pMonitor->Tag + 1;
    // Save contents of log and clear the list
    pMonitor->SaveToFile(LogFileName);
    pMonitor->TraceList->Clear();

}
```

Note: If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's SetTraceCallbackEvent method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CBInfo*:

- *CallType* is reserved for future use.
- *CBInfo* is a pointer to a record that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRType*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

Warning: Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

Using client datasets

Using Client Datasets: Overview

Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by midaslib.dcu or midas.dll. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to

- Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in Using a client dataset with file-based data.
- Cache updates for data from a database server. Client dataset features that support cached updates are described in Using a client dataset to cache updates.
- Represent the data in the client portion of a multi-tiered application. To function in this way, the client dataset must work with an external provider, as described in Using a client dataset with a provider. For information about multi-tiered database applications, see Creating multi-tiered applications.
- Represent the data from a source other than a dataset. Because a client dataset can use the data from an external provider, specialized providers can adapt a variety of information sources to work with client datasets. For example, you can use an XML provider to enable a client dataset to represent the information in an XML document.

Whether you use client datasets for file-based data, caching updates, data from an external provider (such as working with an XML document or in a multi-tiered application), or a combination of these approaches such as a "briefcase model" application, you can take advantage of broad range of features client datasets support for working with data.

Working with Data Using a Client Dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Using data controls for information on how to display database information in data-aware controls.

Client datasets implement all the properties an methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Understanding datasets.

In addition, client datasets implement many of the features common to table type datasets such as

- Sorting records with indexes.
- Using Indexes to search for records.
- Limiting records with ranges.
- Creating master/detail relationships.
- Controlling read/write access
- Creating the underlying dataset

- Emptying the dataset
- Synchronizing client datasets

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for some database functions can involve additional capabilities or considerations. The following topics describe some of these common functions and the differences introduced by client datasets:

- Navigating data
- Limiting What Records Appear
- Editing data .
- Constraining data values
- Sorting and indexing .
- Representing calculated values .
- Copying data from another dataset.
- Adding application-specific information to the data .

Navigating Data in Client Datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset's records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see Navigating datasets.

Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the RecNo property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

Limiting What Records Appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see Displaying and editing a subset of data using filters. For more information about ranges, see Limiting records with ranges.

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than that of other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields.

The various operators and functions that client datasets can use in filters, along with a comparison to other datasets that support filters, is given below:

Filter support in client datasets

Operator or function	Example	Supported by other datasets	Comment
Comparisons			
=	State = 'CA'	Yes	
<>	State <> 'CA'	Yes	
>=	DateEntered >= '1/1/1998'	Yes	

<=	Total <= 100,000	Yes	
>	Percentile > 50	Yes	
<	Field1 < Field2	Yes	
BLANK	State <> 'CA' or State = BLANK	Yes	Blank records do not appear unless explicitly included in the filter.
IS NULL	Field1 IS NULL	No	
IS NOT NULL	Field1 IS NOT NULL	No	
Logical operators			
and	State = 'CA' and Country = 'US'	Yes	
or	State = 'CA' or State = 'MA'	Yes	
not	not (State = 'CA')	Yes	
Arithmetic operators			
+	Total + 5 > 100	Depends on driver	Applies to numbers, strings, or date (time) + number.
-	Field1 - 7 <> 10	Depends on driver	Applies to numbers, dates, or date (time) - number.
*	Discount * 100 > 20	Depends on driver	Applies to numbers only.
/	Discount > Total / 5	Depends on driver	Applies to numbers only.
String functions			
Upper	Upper(Field1) = 'ALWAYS'	No	
Lower	Lower(Field1 + Field2) = 'josp'	No	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	No	Value goes from position of second argument to end or number of chars in third argument. First char has position 1.
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	No	Removes third argument from front and back. If no third argument, trims spaces.
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '$') <> ''	No	See Trim.
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> ''	No	See Trim.
DateTime functions			
Year	Year(DateField) = 2000	No	
Month	Month(DateField) <> 12	No	
Day	Day(DateField) = 1	No	
Hour	Hour(DateField) < 16	No	
Minute	Minute(DateField) = 0	No	
Second	Second(DateField) = 30	No	
GetDate	GetDate - DateField > 7	No	Represents current date and time.

Date	DateField = Date(GetDate)	No	Returns the date portion of a datetime value.
Time	TimeField > Time(GetDate)	No	Returns the time portion of a datetime value.
Miscellaneous			
Like	Memo LIKE '%filters%'	No	Works like SQL-92 without the ESC clause. When applied to BLOB fields, FilterOptions determines whether case is considered.
In	Day(DateField) in (1,7)	No	Works like SQL-92. Second argument is a list of values all with the same type.
*	State = 'M*'	Yes	Wildcard for partial comparisons.

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset.

Note: When fetching data from a provider, you can also limit the data that the client dataset stores by supplying parameters to the provider. For details, see Limiting Records with Parameters.

Editing Data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's Data property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the Delta property. Using a change log serves two purposes:

- The change log is required for applying updates to a database server or external provider component.
- The change log provides sophisticated support for undoing changes.

The LogChanges property lets you disable logging. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications if you do not want the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

Note: Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

Undoing Changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call UndoLastChange. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the

cursor on the current record (*False*). If there are several changes to a record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure. If the removal occurs, *UndoLastChange* returns *True*. Use the ChangeCount property to check whether there are more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.

- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call RevertRecord. *RevertRecord* removes any changes to the current record from the change log.
- To restore a deleted record, first set the StatusFilter property to [*usDeleted*], which makes the deleted records "visible." Next, navigate to the record you want to restore and call *RevertRecord*. Finally, restore the *StatusFilter* property to [*usModified*, *usInserted*, *usUnmodified*] so that the edited version of the dataset (now containing the restored record) is again visible.
- At any point during edits, you can save the current state of the change log using the SavePoint property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling CancelUpdates. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

Saving Changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. Merging changes into data describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to cache updates or to represent the data from an external provider component. The information in the change log is required for resolving updated records with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which attempts to write the modifications to the database server or source dataset, and updates the *Data* property only when the modifications have been successfully committed. See Applying updates for more information about this process.

Constraining Data Values

Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset.

In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider component, the provider can control whether those constraints are sent to the client dataset, and the client dataset can control whether it uses them. For details on how the provider controls whether constraints are included in data packets, see Handling server constraints. For details on how and why client dataset can turn off enforcement of server constraints, see Handling constraints from the server.

Specifying custom constraints

You can use the properties of the client dataset's field components to impose your own constraints on what data users can enter. Each field component has two properties that you can use to specify constraints:

- The DefaultExpression property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is applied back to the database server or source dataset.
- The CustomConstraint property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see Creating a custom constraint.

In addition, you can create record-level constraints using the client dataset's Constraints property. *Constraints* is a collection of TCheckConstraint objects, where each object represents a separate condition. Use the CustomConstraint property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

Sorting and Indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application set up relationships with other datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets:

- Adding a new index
- Deleting and switching indexes
- Using indexes to group data

Note: You may also want to review the material on indexes in table type datasets, which also applies to client datasets. This material is in Using Indexes to search for records and Limiting records with ranges.

Adding a New Index

There are three ways to add indexes to a client dataset:

Methods	Description
Use the IndexFieldNames property	To create a temporary index at runtime that sorts the records in the client dataset, you can use the IndexFieldNames property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index. This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.
Call AddIndex	To create an index at runtime that can be used for grouping, call AddIndex. *AddIndex* lets you specify the properties of the index, including: The name of the index. This can be used for switching indexes at runtime. The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields. How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can set options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order. The default level of grouping support for the index. Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can"t call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.
Use the IndexDefs property	The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call CreateDataSet. See Creating and deleting tables for more information about creating client datasets. As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

Tip: You can index and sort on internally calculated fields with client datasets.

Deleting and Switching Indexes

To remove an index you created for a client dataset, call DeleteIndex and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index when more than one index is available, use the IndexName property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the **Object Inspector**.

Using Indexes to Group Data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to display a field value only if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the GetGroupState method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

Representing Calculated Values

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates.

Using Internally Calculated Fields in Client Datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an OnCalcFields event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnC*alcFields event handler. However, you can optimize your event handler by checking the State property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. Depending on whether you use persistent fields or field definitions, you do this in one of the following ways:

- If you use persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor.
- If you use field definitions, set the InternalCalcField property of the relevant field definition to *True*.

Note: Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

Using Maintained Aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

The following topics describe how to

- Specify aggregates.
- Aggregate Over Groups of Records.
- Obtain aggregate values.

Specifying Aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the Aggregates property. *Aggregates* is a collection of aggregate specifications (TAggregate). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the Add method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note: When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields.

For each aggregate, the Expression property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in the following table

Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

`Sum(Qty * Price)`	`{legal -- summary of an expression on fields }`
`Max(Field1) - Max(Field2)`	`{legal -- expression on summaries }`
`Avg(DiscountRate) * 100`	`{legal -- expression of summary and constant }`
`Min(Sum(Field1))`	`{illegal -- nested summaries }`
`Count(Field1) - Field2`	`{illegal -- expression of summary and field }`

Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value. Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
[Delphi]
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

```
[C++]
Agg->Expression = "Sum(Amount)";
Agg->IndexName = "SalesCust";
Agg->GroupingLevel = 1;
Agg->AggregateName = "Total for Rep";
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

Obtaining Aggregate Values

To get the value of a maintained aggregate, call the Value method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the GetGroupState method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's Aggregates is automatically updated to include the appropriate aggregate specification. The AggFields property contains the new aggregated field component, and the FindField method returns it.

Copying Data from Another Dataset

To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Assigning Data Directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
[Delphi]
ClientDataSet1.Data := ClientDataSet2.Data;
```

```
[C++]
ClientDataSet1->Data = ClientDataSet2->Data;
```

Note: When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
[Delphi]
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

```
[C++]
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SourceDataSet;
ClientDataSet1->Data = TempProvider->Data;
delete TempProvider;
```

Note: When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
[Delphi]
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

```
[C++]
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = ClientDataSet1;
TempProvider->ApplyUpdates(SourceDataSet->Delta, -1, ErrCount);
delete TempProvider;
```

Cloning a Client Dataset Cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

Adding Application-specific Information to the Data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the SetOptionalParam method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the GetOptionalParam method, passing in the name that was used when the information was stored.

Using a Client Dataset to Cache Updates

By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the database, your application is notified immediately: the dataset raises an exception when you post the record.

If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network traffic, you may want to cache updates locally. When you cache updates, you application retrieves data from the database, caches and edits it locally, and then applies the cached updates to the database in a single transaction. When you cache updates, changes to a dataset (such as posting changes or deleting records) are stored locally instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate

for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

Although the BDE and ADO provide alternate mechanisms for caching updates, using a client dataset for caching updates has several advantages:

- Applying updates when datasets are linked in master/detail relationships is handled for you. This ensures that updates to multiple linked datasets are applied in the correct order.
- Client datasets give you the maximum of control over the update process. You can set properties to influence the SQL that is generated for updating records, specify the table to use when updating records from a multi-table join, or even apply updates manually from a *BeforeUpdateRecord* event handler.
- When errors occur applying cached updates to the database server, only client datasets (and dataset providers) provide you with information about the current record value on the database server in addition to the original (unedited) value from your dataset and the new (edited) value of the update that failed.
- Client datasets let you specify the number of update errors you want to tolerate before the entire update is rolled back.

The following topics describe in more detail on how to use a client dataset to cache updates:

- Overview of using cached updates.
- Choosing the type of dataset for caching updates.
- Indicating what records are modified.
- Updating records.

Overview of Using Cached Updates

To use cached updates, the following order of processes must occur in an application:

Indicate the data you want to edit. How you do this depends on the type of client dataset you are using:

- If you are using *TClientDataSet*, Specify the provider component that represent the data you want to edit.
- If you are using a client dataset associated with a particular data access mechanism, you must
- Identify the database server by setting the *DBConnection* property to an appropriate connection component.
- Indicate what data you want to see by specifying the CommandText and *CommandType* properties. *CommandType* indicates whether *CommandText* is an SQL statement to execute, the name of a stored procedure, or the name of a table. If *CommandText* is a query or stored procedure, use the Params property to provide any input parameters.
- Optionally, use the Options property to indicate whether nested detail sets and BLOB data should be included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) should be disabled, whether a single update can affect multiple server records, and whether the client dataset's records are refreshed when it applies updates. *Options* is identical to a provider's *Options* property. As a result, it allows you to set options that are not relevant or appropriate. For example, there is no reason to include *poIncFieldProps*, because the client dataset does not fetch its data from a dataset with persistent fields. Conversely, you do not want to exclude *poAllowCommandText*, which is included by default, because that would disable the *CommandText* property, which the client dataset uses to specify what data it wants. For information on the provider's *Options* property, see Setting options that influence the data packets.

Display and edit the data, permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory.This process is described in Editing data.

Fetch additional records as necessary. By default, client datasets fetch all records and store them in memory. If a dataset contains many records or records with large BLOB fields, you may want to change this so that the client

dataset fetches only enough records for display and re-fetches as needed. For details on how to control the record-fetching process, see Requesting data from the source dataset or document.

Optionally, refresh the records. As time passes, other users may modify the data on the database server. This can cause the client dataset's data to deviate more and more from the data on the server, increasing the chance of errors when you apply updates. To mitigate this problem, you can refresh records that have not already been edited. See Refreshing records for details.

Apply the locally cached records to the database or cancel the updates. For each record written to the database, a *BeforeUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see Updating records.

Instead of applying updates, an application can cancel the updates, emptying the change log without writing the changes to the database. You can cancel the updates by calling CancelUpdates method. All deleted records in the cache are undeleted, modified records revert to original values, and newly inserted record simply disappear.

Choosing the Type of Dataset for Caching Updates

Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism. These are listed in the following table:

Specialized client datasets for caching updates

Client dataset	Data access mechanism
TBDEClientDataSet	Borland Database Engine
TSimpleDataSet	dbExpress
TIBClientDataSet	InterBase Express

In addition, you can cache updates using the generic client dataset (TClientDataSet) with an external provider and source dataset. For information about using *TClientDataSet* with an external provider, see Using a client dataset with a provider.

Note: The specialized client datasets associated with each data access mechanism actually use a provider and source dataset as well. However, both the provider and the source dataset are internal to the client dataset.

It is simplest to use one of the specialized client datasets to cache updates. However, there are times when it is preferable to use *TClientDataSet* with an external provider:

- If you are using a data access mechanism that does not have a specialized client dataset, you must use *TClientDataSet* with an external provider component. For example, if the data comes from an XML document or custom dataset.
- If you are working with tables that are related in a master/detail relationship, you should use *TClientDataSet* and connect it, using a provider, to the master table of two source datasets linked in a master/detail relationship. The client dataset sees the detail dataset as a nested dataset field. This approach is necessary so that updates to master and detail tables can be applied in the correct order.
- If you want to code event handlers that respond to the communication between the client dataset and the provider (for example, before and after the client dataset fetches records from the provider), you must use *TClientDataSet* with an external provider component. The specialized client datasets publish the most important events for applying updates (*OnReconcileError*, *BeforeUpdateRecord* and OnGetTableName), but do not publish the events surrounding communication between the client dataset and its provider, because they are intended primarily for multi-tiered applications.
- When using the BDE, you may want to use an external provider and source dataset if you need to use an update object. Although it is possible to code an update object from the *BeforeUpdateRecord* event handler of

TBDEClientDataSet, it can be simpler just to assign the *UpdateObject* property of the source dataset. For information about using update objects, see Using update objects to update a dataset.

Indicating What Records Are Modified

While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an "Undo" button.

The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred:

UpdateStatus indicates what type of update, if any, has occurred for the current record. It can be any of the following values:

- *usUnmodified* indicates that the current record is unchanged.
- *usModified* indicates that the current record has been edited.
- *usInserted* indicates a record that was inserted by the user.
- *usDeleted* indicates a record that was deleted by the user.

StatusFilter controls what type of updates in the change log are visible. *StatusFilter* works on cached records in much the same way as filters work on regular data. *StatusFilter* is a set, so it can contain any combination of the following values:

- *usUnmodified* indicates an unmodified record.
- *usModified* indicates a modified record.
- *usInserted* indicates an inserted record.
- *usDeleted* indicates a deleted record.

By default, *StatusFilter* is the set [*usModified*, *usInserted*, *usUnmodified*]. You can add *usDeleted* to this set to provide feedback about deleted records as well.

Note: *UpdateStatus* and *StatusFilter* are also useful in *BeforeUpdateRecord* and *OnReconcileError* event handlers. For information about *BeforeUpdateRecord*, see Intervening as updates are applied For information about *OnReconcileError*, see Reconciling Update Errors.

The following example shows how to provide feedback about the update status of records using the *UpdateStatus* method. It assumes that you have changed the *StatusFilter* property to include *usDeleted*, allowing deleted records to remain visible in the dataset. It further assumes that you have added a calculated field to the dataset called "Status. "

```
[Delphi]
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
```

```
[C++]
void __fastcall TForm1::ClientDataSet1CalcFields(TDataSet *DataSet)
{
  switch (DataSet->UpdateStatus())
  {
    case usUnmodified:
      ClientDataSet1Status->Value = NULL; break;
    case usModified:
      ClientDataSet1Status->Value = "M"; break;
    case usInserted:
      ClientDataSet1Status->Value = "I"; break;
    case usDeleted:
      ClientDataSet1Status->Value = "D"; break;
  }
}
```

Updating Records

The contents of the change log are stored as a data packet in the client dataset's *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document).

When a client applies updates to the server, the following steps occur:

1. The client application calls the ApplyUpdates method of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The provider applies the updates, caching any problem records that it can't resolve itself. See Responding to client update requests for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

Applying Updates

Changes made to the client dataset's local copy of data are not sent to the database server (or XML document) until the client application calls the ApplyUpdates method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called Delta) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.)

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

- It indirectly calls the provider's ApplyUpdates method. The provider's *ApplyUpdates* method writes the updates to the database, source dataset, or XML document and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
- The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the Reconcile method. *Reconcile* is an error-handling routine that calls the OnReconcileError event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see Reconciling Update Errors.
- Finally, *Reconcile* removes successfully applied changes from the change log and updates Data to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

Warning: In some cases, the provider can't determine how to apply updates (for example, when applying updates from a stored procedure or multi-table join). Client datasets and provider components generate events that let you handle these situations. See Intervening as updates are applied for details.

Tip: If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. *TClientDataSet* receives a BeforeApplyUpdates event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), *TClientDataSet* receives an AfterApplyUpdates event where you can respond to any persistent state information returned by the application server.

Intervening as Updates Are Applied

When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in Responding to client update requests.

When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and two events that let you influence how the internal provider applies updates.

- UpdateMode controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see Influencing how updates are applied.
- OnGetTableName lets you supply the provider with the name of the database table to which it should apply updates. This lets the provider generate the SQL statements for updates when it can't identify the database table from the stored procedure or query specified by *CommandText*. For example, if the query executes a multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates. An *OnGetTableName* event handler has three parameters: the internal provider component, the internal dataset that fetched the data from the server, and a parameter to return the table name to use in the generated SQL.
- BeforeUpdateRecord occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.) A *BeforeUpdateRecord* event handler has five parameters: the internal provider component, the internal dataset that fetched the data from the server, a delta packet that is positioned on the record that is about to be updated, an indication of whether the update is an insertion, deletion, or modification, and a parameter that returns whether the event handler performed the update. The use of these is illustrated in the following event handler. For simplicity, the example assumes the SQL statements are available as global variables that only need field values:

```
[Delphi]
procedure TForm1.SimpleDataSet1BeforeUpdateRecord(Sender: TObject;
   SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
   var Applied Boolean);
var
  SQL: string;
  Connection: TSQLConnection;
begin
  Connection := (SourceDS as TSimpleDataSet).Connection;
  case UpdateKind of
  ukModify:
    begin
     { 1st dataset: update Fields[1], use Fields[0] in where clause }
      SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: update Fields[2], use Fields[3] in where clause }
      SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukDelete:
    begin
     { 1st dataset: use Fields[0] in where clause }
      SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: use Fields[3] in where clause }
      SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukInsert:
    begin
     { 1st dataset: values in Fields[0] and Fields[1] }
       SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: values in Fields[2] and Fields[3] }
      SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  end;
  Applied := True;
end;
```

```
[C++]
void __fastcall TForm1::SimpleDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TSQLConnection *pConn := (dynamic_cast<TSimpleDataSet *>(SourceDS)->Connection);
  char buffer[256];
  switch (UpdateKind)
  case ukModify:
    // 1st dataset: update Fields[1], use Fields[0] in where clause
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[1]->NewValue,
            DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: update Fields[2], use Fields[3] in where clause
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukDelete:
```

```
    // 1st dataset: use Fields[0] in where clause
    sprintf(buffer, DeleteStmt1, DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: use Fields[3] in where clause
    sprintf(buffer, DeleteStmt2, DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukInsert:
    // 1st dataset: values in Fields[0] and Fields[1]
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[0]->NewValue,
            DeltaDS->Fields->Fields[1]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: values in Fields[2] and Fields[3]
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
}
```

Reconciling Update Errors

There are two events that let you handle errors that occur during the update process:

- During the update process, the internal provider generates an OnUpdateError event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead.
- After the entire update operation is finished, the client dataset generates an OnReconcileError event for every record that the provider could not apply to the database server.

You should always code an *OnReconcileError* or *OnUpdateError* event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

DataSet: A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the CurValue, OldValue, and NewValue properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.

E: An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.

UpdateKind: The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).

Action: A **var** parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to

- Skip this record, leaving it in the change log. (rrSkip or raSkip)
- Stop the entire reconcile operation. (rrAbort or raAbort)
- Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
- Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)

- Ignore the error completely. (rrIgnore) This possibility only exists in the *OnUpdateError* event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into *Data*, as if the provider had applied the update.
- Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the *OnReconcileError* event handler.
- Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the *OnReconcileError* event handler.

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the objrepos directory. (To use this dialog, add RecError to your uses clause.)

```
[Delphi]
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

```
[C++]
void __fastcall TForm1::ClientDataSetReconcileError(TCustomClientDataSet *DataSet,
    EReconcileError *E, TUpdateKind UpdateKind, TReconcileAction &Action)
{
  Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

Using a Client Dataset with a Provider

A client dataset uses a provider to supply it with data and apply updates when

- It caches updates from a database server or another dataset.
- It represents the data in an XML document.
- It stores the data in the client portion of a multi-tiered application.

For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data.

An external provider component can reside in the same application as the client dataset, or it can be part of a separate application running on another system. For more information about provider components, see Using Provider Components. For more information about applications where the provider is in a separate application on another system, see Creating multi-tiered applications.

When using an (internal or external) provider, the client dataset always caches any updates. For information on how this works, see Using a client dataset to cache updates.

The following topics describe additional properties and methods of the client dataset that enable it to work with a provider:

- Specifying a provider
- Requesting data from the source dataset or document.
- Getting Parameters From the Application Server
- Passing parameters to the source dataset
- Handling constraints from the server

- Refreshing records.
- Communicating with providers using custom events
- Overriding the source dataset

Specifying a Provider

Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component.

The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

Provider's location	How to associate TClientDataSet
The provider is in the same application as the client dataset	If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the ProviderName property in the **Object Inspector**. This works as long as the provider has the same Owner as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's SetProvider method If you think you may eventually scale up to a remote provider, or if you want to make calls directly to the *IAppServer* interface, you can also set the ConnectionBroker property to a TLocalConnection component. If you use *TLocalConnection*, the *TLocalConnection* instance manages the list of all providers that are local to the application, and handles the client dataset's *IAppServer* calls. If you do not use *TLocalConnection*, the application creates a hidden object that handles the *IAppServer* calls from the client dataset.
The provider is on a remote application server	If the provider is on a remote application server, then, in addition to the ProviderName property, you need to specify a component that connects the client dataset to the application server. There are two properties that can handle this task: RemoteServer, which specifies the name of a connection component from which to get a list of providers, or ConnectionBroker, which specifies a centralized broker that provides an additional level of indirection between the client dataset and the connection component. The connection component and, if used, the connection broker, reside in the same data module as the client dataset. The connection component establishes and maintains a connection to an application server, sometimes called a "data broker." For more information, see The structure of the client application At design time, after you specify *RemoteServer* or *ConnectionBroker,* you can select a provider from the drop-down list for the *ProviderName* property in the **Object Inspector**. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

Note: If the connection component is an instance of *TDCOMConnection*, the application server must be registered on the client machine.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

Requesting Data from the Source Dataset or Document

Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet, TSimpleDataSet,* and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*.

You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.

Incremental fetching

By changing the PacketRecords property, you can specify that the client dataset fetches data in smaller chunks. *PacketRecords* specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
[Delphi]
ClientDataSet1.PacketRecords := 10;
```

```
[C++]
ClientDataSet1->PacketRecords = 10;
```

This process of fetching records in batches is called "incremental fetching". Client datasets use incremental fetching when *PacketRecords* is greater than zero.

To fetch each batch of records, the client dataset calls GetNextPacket. Newly fetched packets are appended to the end of the data already in the client dataset. *GetNextPacket* returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning: Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server. See Supporting state information in remote data modules for information on how to use incremental fetching with stateless remote data modules.

Note: You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

Fetch-on-demand

Automatic fetching of records is controlled by the FetchOnDemand property. When *FetchOnDemand* is *True* (the default), the client dataset automatically fetches records as needed. To prevent automatic fetching of records, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *False*, the application must explicitly call *GetNextPacket* to fetch records.

For example, Applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client

dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

The provider controls whether the records in data packets include BLOB data and nested detail datasets. If the provider excludes this information from records, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is *False,* and the provider does not include BLOB data and detail datasets with records, you must explicitly call the FetchBlobs or FetchDetails method to retrieve this information.

Getting Parameters from the Application Server

There are two circumstances when the client dataset needs to fetch parameter values:

- The application needs the value of output parameters on a stored procedure.
- The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

Client datasets store parameter values in their Params property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset. However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the FetchParams method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note: There is never a need to call *FetchParams* when the client dataset uses an internal provider and source dataset, because the *Params* property always reflects the parameters of the internal source dataset. With *TClientDataSet*, the *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a table type dataset, there are no parameters to fetch.

The *Params* property can also be used to pass parameter values to the source dataset. For details on how to do this, see Passing parameters to the source dataset.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the Execute method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

Passing Parameters to the Source Dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Input parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the Params property in the **Object Inspector**. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the **Object Inspector** to edit the properties of that parameter.

At runtime, use the CreateParam method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code adds an input parameter named CustNo with a value of 605:

```
[Delphi]
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
AsInteger := 605;
```

```
[C++]
TParam *pParam = ClientDataSet1->Params->CreateParam(ftInteger, "CustNo", ptInput);
pParam->AsInteger = 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

Note: You may want to initialize parameter values from the current settings on the source dataset. You can do this by right-clicking the client dataset and choosing Fetch Params at design time or calling the FetchParams method at runtime.

Sending Query or Stored Procedure Parameters

When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the Params property to specify parameter values. When the client dataset requests data from the source dataset or uses its Execute method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note: Parameter names should match the names of the corresponding parameters on the source dataset.

Limiting Records with Parameters

If the client dataset is

- a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component
- a *TSimpleDataSet or a TBDEClientDataSet* instance whose *CommandType* property is *ctTable*

then it can use the Params property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable.

Each parameter name must match the name of a field. When using *TClientDataSet*, these are the names of fields in the *TTable* or *TSQLTable* component associated with the provider. When using *TSimpleDataSet* or *TBDEClientDataSet*, these are the names of fields in the table on the database server. The data in the client dataset then includes only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the

records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

Handling Constraints from the Server

When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process.

Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in Constraining data values.

It is more convenient, however, if the server constraints are automatically included in data packets. Then you need not explicitly specify default expressions and constraints, and the client dataset changes the values it enforces when the server constraints change. By default, this is exactly what happens: if the source dataset is aware of server constraints, the provider automatically includes them in data packets and the client dataset enforces them when the user posts edits to the change log.

Note: Only datasets that use the BDE can import constraints from the server. This means that server constraints are only included in data packets when using *TBDEClientDataSet* or *TClientDataSet* with a provider that represents a BDE-based dataset. For more information on how to import server constraints and how to prevent a provider from including them in data packets, see Handling server constraints.

Note: For more information on working with the constraints once they have been imported, see Using server constraints.

While importing server constraints and expressions is an extremely valuable feature that helps an application preserve data integrity, there may be times when it needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value of a field, but the client dataset uses incremental fetching, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call the DisableConstraints method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset's EnableConstraints method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip: Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

Refreshing Records

Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data.

Like any other dataset, client datasets have a Refresh method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client datasets can also update the data while leaving the change log intact. To do this, call the RefreshRecord method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

Warning: It is not always appropriate to call *RefreshRecord*. If the user's edits conflict with changes made to the underlying dataset by other users, calling *RefreshRecord* masks this conflict. When the client dataset applies its updates, no reconcile error occurs and the application can't resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following *AfterScroll* refreshes the current record every time the user moves to a new record (ensuring the most up-to-date value), but only when it is safe to do so.:

```
[Delphi]
procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
if ClientDataSet1.UpdateStatus = usUnModified then
ClientDataSet1.RefreshRecord;
end;
```

```
[C++]
void __fastcall TForm1::ClientDataSet1AfterScroll(TDataSet *DataSet)
{
  if (ClientDataSet1->UpdateStatus == usUnModified)
    ClientDataSet1->RefreshRecord();
}
```

Communicating with Providers Using Custom Events

Client datasets communicate with a provider component through a special interface called IAppServer. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server, or (in the case of a SOAP server) an interface generated by the connection component.

TClientDataSet provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1. The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2. The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
3. The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4. The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
5. The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between client dataset and provider.

In addition, the client dataset has a special method, DataRequest, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an is the OnDataRequest event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Overriding the Dataset On the Application Server

The client datasets that are associated with a particular data access mechanism use the CommandText and *Com*mandType properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure.

If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's CommandText property replaces the property on the provider's dataset that specifies what data it represents. This allows *TClientDataSet* to specify dynamically what data it wants to see.

By default, external provider components do not let client datasets use the *CommandText* value in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the Options property of the provider. Otherwise, the value of *CommandText* is ignored.

Note: Never remove *poAllowCommandText* from the *Options* property of *TBDEClientDataSet* or *TIBClientDataSet*. The client dataset's *Options* property is forwarded to the internal provider, so removing *poAllowCommandText* prevents the client dataset from specifying what data to access.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an Execute command to provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the AppServer property. This property represents the interface through which the client dataset communicates with its provider.

Using a Client Dataset with File-based Data

Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications and "briefcase model" applications. The special files that client datasets use for their data are called MyBase.

Tip: All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead.

In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables
- Load saved data
- Merge edits into its data
- Save data

Creating a New Dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table type dataset. See Creating and deleting tables for details.
- You can copy an existing dataset (at design or runtime).
- You can create a client dataset from an arbitrary XML document. See Converting XML documents into data packets for details.

Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you start with the metadata for your client dataset already defined, making it easier to set up the user interface.

Loading Data from a File or Stream

To load data from a file, call a client dataset's LoadFromFile method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the FileName property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile* (SaveToStream) method, or generated from an XML document. For more information about saving data to a file or stream, see Saving data to a file or stream. For information about creating client dataset data from an XML document, see Using XML in database applications.

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property. However, the only indexes that are read from the file are those that were created with the dataset.

Merging Changes into Data

When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the MergeChangeLog method. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, Data contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the ChangeCount property to *0*.

Warning: Do not call *MergeChangeLog* for client datasets that use a provider. In this case, call *ApplyUpdates* to write changes to the database. For more information, see Applying updates.

Note: It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see Assigning data directly.

If you do not want to use the extended undo capabilities of the change log, you can set the client dataset's LogChanges property to *False*. When *LogChanges* is *False*, edits are automatically merged when you post records and there is no need to call *MergeChangeLog*.

Saving Data to a File or Stream

Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the SaveToFile method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

Note: *SaveToFile* does not preserve any indexes you added to the client dataset at runtime, only indexes that were added when you created the client dataset.

If you always save the data to the same file, you can use the FileNameproperty instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the SaveToStream method. *SaveToStream* takes one parameter, a stream object that receives the data.

Note: If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.

Using a Simple Dataset

TSimpleDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

TSimpleDataSet works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSimpleDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset.

Using a simple dataset can simplify the process of two-tiered application development because you don't need to work with as many components.

When to use TSimpleDataSet provides information on when and how to use a simple dataset:

When to Use TSimpleDataSet

TSimpleDataSet is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications.

There are times, however, when it is more appropriate to use *TClientDataSet*:

- If you are not using data from a database server (for example, if you are using a dedicated file on disk), then *TClientDataSet* has the advantage of less overhead.
- Only *TClientDataSet* can be used in a multi-tiered database application . Thus, if you are writing a multi-tiered application, or if you intend to scale up to a multi-tiered application eventually, you should use *TClientDataSet* with an external provider and source dataset.

- Because the source dataset is internal to the simple dataset component, you can't link two source datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two simple datasets into a master/detail relationship.)
- The simple dataset does not surface any of the events or properties that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Setting up a simple dataset provides information on setting up a simple dataset:

Setting Up a Simple Dataset

Setting up a simple dataset requires two essential steps. Set up:

1. The connection information.
2. The dataset information.

The following steps describe setting up a simple dataset in more detail.

To use TSimpleDataSet:

1. Place the *TSimpleDataSet* component in a data module or on a form. Set its Name property to a unique value appropriate to your application.
2. Identify the database server that contains the data. There are two ways to do this:
   - If you have a named connection in the connections file, expand the *Connection* property and specify the ConnectionName value.
   - For greater control over connection properties, transaction support, login support, and the ability to use a single connection for more than one dataset, use a separate *TSQLConnection* component instead. Specify the *TSQLConnection* component as the value of the Connection property. For details on *TSQLConnection*, see Connecting to databases.
3. To indicate what data you want to fetch from the server, expand the DataSet property and set the appropriate values. There are three ways to fetch data from the server:
   - Set CommandType to *ctQuery* and set CommandText to an SQL statement you want to execute on the server. This statement is typically a SELECT statement. Supply the values for any parameters using the Params property.
   - Set CommandType to *ctStoredProc* and set CommandText to the name of the stored procedure you want to execute. Supply the values for any input parameters using the Params property.
   - Set CommandType to *ctTable* and set CommandText to the name of the database tables whose records you want to use.
4. If the data is to be used with visual data controls, add a data source component to the form or data module, and set its *DataSet* property to the *TSimpleDataSet* object. The data source component forwards the data in the client dataset's in-memory cache to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Activate the dataset by setting the *Active* property to *true* (or, at runtime, calling the *Open* method).
6. If you executed a stored procedure, use the Params property to retrieve any output parameters.

7 When the user has edited the data in the simple dataset, you can apply those edits back to the database server by calling the ApplyUpdates method. Resolve any update errors in an OnReconcileError event handler. For more information on applying updates, see Updating records.

Using provider components

Using Provider Components

Provider components (TDataSetProvider and TXMLTransformProvider) supply the most common mechanism by which client datasets obtain their data. Providers

- Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called "providing."
- Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called "resolving."

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in a dataset or XML document or to apply updates. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

When using *TBDEClientDataSet*, *TSimpleDataSet*, or *TIBClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet* or *TXMLBroker*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. The client datasets that have internal providers surface some of the internal provider's properties and events as their own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset (or XML broker), or it can reside on an application server as part of a multi-tiered application.

The following topics describe how to use a provider component to control the interaction with client datasets or XML brokers.

- Determining the Source of Data
- Communicating with the Client Dataset
- Choosing How to Apply Updates Using a Dataset Provider
- Controlling what Information is Included in Data Packets
- Responding to Client Data Requests
- Responding to Client Update Requests
- Responding to Client-generated Events

- Handling Server Constraints

Determining the Source of Data

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following:

- To provide the data from a dataset, use TDataSetProvider.
- To provide the data from an XML document, use TXMLTransformProvider.

Using a dataset as the source of the data

If the provider is a dataset provider (*TDataSetProvider*), set the DataSet property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the **Object Inspector**.

TDataSetProvider interacts with the source dataset using the IProviderSupport interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions.

The dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, *dbExpress* datasets, and InterBase Express datasets) override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets don't add anything to the inherited *IProviderSupport* implementation, but can still be used as a source dataset as long as the *ResolveToDataSet* property of the provider is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Using an XML document as the source of the data

If the provider is an XML provider, set the XMLDataFile property of the provider to indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you must also specify how to transform that document into data packets. This transformation is handled by the provider's TransformRead property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation. For more information on using XML providers, see Using an XML document as the source for a provider.

Communicating with the Client Dataset

All communication between a provider and a client dataset or XML broker takes place through an IAppServer interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a TLocalConnection component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module or (in the case of a SOAP server) an interface generated by the connection component.

Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset or XML broker. However, when necessary, you can make direct calls to the *IAppServer* interface by using the AppServer property of a client dataset.

The following table lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-

tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client dataset and a provider to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

AppServer interface members

IAppServer	Provider Component	TClientDataSet
AS_ApplyUpdatesAS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event.
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method.
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event.
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event.
AS_GetProviderNames method	Used to identify all available providers.	Used to create a design-time list for ProviderName property.
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

Choosing How to Apply Updates Using a Dataset Provider

TXMLTransformProvider components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component.

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the ResolveToDataSet property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

Controlling What Information Is Included in Data Packets

When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets

- Adding custom information to data packets

Note: These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

Specifying What Fields Appear in Data Packets

When using a dataset provider, you can control what fields are included in data packets by creating persistent fields on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields.

If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only to ensure uniqueness, set the ProviderFlags property for those fields to include *pfHidden*.

Note: Including enough fields to avoid duplicate records is also a consideration when the provider's source dataset represents a query. You must specify the query so that it includes enough fields to ensure all records are unique, even if your application does not use all the fields.

Setting Options That Influence the Data Packets

The Options property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Provider options

Value	Meaning
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises exception. (This does not affect the client dataset's ability to insert or delete records).
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data)
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records)
poFetchBlobsOnDemand	BLOB field values are not included in data packets. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider's dataset represents the master of a master/detail relationship, nested detail values are not included in data packets. Instead, client datasets request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details.

poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues*.
poCascadeDeletes	When the provider's dataset represents the master of a master/detail relationship, the server automatically deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.
poCascadeUpdates	When the provider's dataset represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents.
poRetainServerOrder	The client dataset should not re-sort the records in the dataset to enforce a default order.

Adding Custom Information to Data Packets

Dataset providers can add application-defined information to data packets using the OnGetDataSetProperties event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their GetOptionalParam method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Add multiple attributes by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only the time the data was provided is returned when client datasets apply updates:

```
[Delphi]
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

```
[C++]
void __fastcall TMyDataModule1::Provider1GetDataSetProperties(TObject *Sender, TDataSet
*DataSet, out OleVariant Properties)
{
  int ArrayBounds[2];
```

```
  ArrayBounds[0] = 0;
  ArrayBounds[1] = 1;
  Properties = VarArrayCreate(ArrayBounds, 1, varVariant);
  Variant values[3];
  values[0] = Variant("TimeProvided");
  values[1] = Variant(Now());
  values[2] = Variant(true);
  Properties[0] = VarArrayOf(values,2);
  values[0] = Variant("TableSize");
  values[1] = Variant(DataSet->RecordCount);
  values[2] = Variant(false);
  Properties[1] = VarArrayOf(values,2);
}
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's OnUpdateData event:

```
[Delphi]
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

```
[C++]
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  Variant WhenProvided = DataSet->GetOptionalParam("TimeProvided");
  ...
}
```

Responding to Client Data Requests

Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling GetRecords (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client, write an OnGetData event handler. *OnGetData* event handlers provide the data packet as a parameter in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client.

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *GetRecords*. This communication takes place using the BeforeGetRecords and AfterGetRecords event handlers.

Responding to Client Update Requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the ApplyUpdates method (indirectly, through the *IAppServer* interface).

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the BeforeApplyUpdates and AfterApplyUpdates event handlers.

If you are using a dataset provider, a number of additional events allow you more control:

When a dataset provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdate*Data event, the provider writes the changes to the database or source dataset.

The provider performs the update on a record-by-record basis. Before the dataset provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or a database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the dataset provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its StatusFilter property.

Note: Applications must supply extra support when the updates are directed at a dataset that does not represent a single table.

Editing Delta Packets Before Updating the Database

Before a dataset provider applies updates to the database, it generates an OnUpdateData event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the UpdateStatus property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in the following table:

UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
[Delphi]

procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

```
[C++]
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  DataSet->First();
  while (!DataSet->Eof)
  {
    if (DataSet->UpdateStatus == usInserted)
    {
      DataSet->Edit();
      DataSet->FieldByName("DateCreated")->AsDateTime = Date();
      DataSet->Post();
    }
    DataSet->Next();
  }
}
```

Influencing How Updates Are Applied

The OnUpdateData event gives your dataset provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the UpdateMode property of the provider. *UpdateMode* can be assigned any of the following values:

UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereKeyOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the ProviderFlags property. *ProviderFlags* is a set that can include any of the values in the following table

ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when *UpdateMode* is *upWhereAll* or *upWhereChanged*.
pfInUpdate	The field appears in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when *UpdateMode* is *upWhereKeyOnly*.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
[Delphi]
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

```
[C++]
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  DataSet->FieldByName("EMPNO")->ProviderFlags.Clear();
  DataSet->FieldByName("EMPNO")->ProviderFlags << pfInWHere << pfInKey;
  DataSet->FieldByName("TITLE")->ProviderFlags.Clear();
  DataSet->FieldByName("TITLE")->ProviderFlags << pfInUpdate;
  DataSet->FieldByName("DEPT")->ProviderFlags.Clear();
  DataSet->FieldByName("DEPT")->ProviderFlags << pfInWhere;
}
```

Note: You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

Screening Individual Updates

Immediately before each update is applied, a dataset provider receives a BeforeUpdateRecord event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

Resolving Update Errors On the Provider

When an error condition arises as the dataset provider tries to post a record in the delta packet, an OnUpdateError event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the NewValue, OldValue, and CurValue properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

Applying Updates to Datasets That do Not Represent a Single Table

When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table type datasets or "live" *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied.

If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider uses the update object. However, if there is no update object, you can supply the table name programmatically in an OnGetTableName event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the BeforeUpdateRecord event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the provider does not generate an error.

Note: If the provider is associated with a BDE-enabled dataset, you can use an update object in the *BeforeUpdateRecord* event handler to apply updates using customized SQL statements.

Responding to Client-generated Events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the OnDataRequest event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the DataRequest method of the client dataset.

Handling Server Constraints

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.
- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

Note: This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications manage. You can take advantage of server constraints in multi-tiered database applications without having to duplicate the constraints in application server or client application code.

If the provider is working with a BDE-enabled dataset, the *Constraints* property lets you replicate and apply server constraints to data passed to and received from client datasets. When *Constraints* is *True* (the default), server constraints stored in the source dataset are included in data packets and affect client attempts to update data.

Warning: Before the provider can pass constraint information on to client datasets, it must retrieve the constraints from the database server.

There may be times when you do not want to apply server constraints to data sent to a client dataset. For example, a client dataset that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the provider to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see Handling constraints from the server.

Creating multi-tiered applications

Creating Multi-tiered Applications: Overview

A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the "three-tiered model," a multi-tiered application is partitioned into thirds:

- **Client application:** provides a user interface on the user's machine.
- **Application server:** resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server:** provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a "data broker." You usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. See Understanding multi-tiered database applications for an overview of this technology and the architecture of a typical three-tiered application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

Building a multi-tiered application provides details on how to apply this architecture to build a three-tiered application. Writing Web-based client applications describes how to combine this architecture with other technologies to create a Web-based multi-tiered application.

Advantages of the Multi-tiered Database Model

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP or COM+, you can take advantage of the security models they support.

Understanding Multi-tiered Database Applications

Multi-tiered applications use the components on the DataSnap page, the Data Access page, and possibly the WebServices page of the **Tool palette**, plus a remote data module that is created by a wizard on the Multitier or WebServices page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application are described in the following table:

Components used in multi-tiered applications

Component	Description
Remote data modules	Specialized data modules that can act as a COM Automation server or implement a Web Service to give client applications access to any providers they contain. Used on the application server.
Provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
Client dataset component	A specialized dataset that uses midas.dll or midaslib.dcu to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
Connection components	A family of components that locate the server, form connections, and make the *IAppServer* interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

The provider and client dataset components require midas.dll or midaslib.dcu, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.dll, you must deploy it on both application server and client application.)

An overview of the architecture into which these components fit is described in Using a multi-tiered architecture. For more information on how these components fit together to create a multi-tiered application, see

- Overview of a Three-tiered Application
- The Structure of the Client Application
- The Structure of the Application Server
- Choosing a Connection Protocol

Overview of a Three-tiered Application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface for communicating with the application server.
2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called "providing."
4. The client decodes the data packet and displays the data to the user.
5. As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
6. Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
7. The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can't be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client's changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called "resolving."
8. When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
9. The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
10. The client refreshes its data from the server.

The Structure of the Client Application

To the end user, the client application of a multi-tiered application looks and behaves no differently than a two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a TClientDataSet component. For detailed information about using the properties, events, and methods of client datasets, see Using Client Datasets.

TClientDataSet fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Using Provider Components. For details about client dataset features that facilitate its communication with a provider, see Using a Client Dataset with a Provider.

The client dataset communicates with the provider through the IAppServer interface. It gets this interface from a connection component. The connection component establishes the connection to the application server. Different connection components are available for using different communications protocols.

These connection components are summarized in the following table:

Connection components

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)

TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP and XML)
TCorbaConnection	CORBA (IIOP)

Note: The DataSnap category of the **Tool palette** also includes a connection component that does not connect to an application server at all, but instead supplies an IAppServer interface for client datasets to use when communicating with providers in the same application. This component, TLocalConnection, is not required, but makes it easier to scale up to a multi-tiered application later.

For more information about using connection components, see Connecting to the Application Server.

The Structure of the Application Server

When you set up and run an application server, it does not establish any connection with client applications. Rather, client applications initiate and maintain the connection. The client application uses a connection component to connect to the application server, and uses the interface of the application server to communicate with a selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface (for application servers that also function as a Web Service, the remote data module supports the *IAppServerSOAP* interface as well, and uses it in preference to *IAppServer*.) Client applications use the remote data module's interface to communicate with providers on the application server. When the remote data module uses *IAppServerSOAP*, the connection component adapts this to an *IAppServer* interface that client datasets can use.

There are three types of remote data modules:

- **TRemoteDataModule**: This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLE to connect to the application server, unless you want to install the application server with COM+.
- **TMTSDataModule**: This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with COM+ (or MTS). You can use MTS remote data modules with DCOM, HTTP, sockets, or OLE. See Using transactional data modules for information about the benefits and limitations of using MTS or COM+ with the application server.
- **TSoapDataModule**: This is a data module that implements an *IAppServerSOAP* interface in a Web Service application. Use this type of remote data module to provide data to clients that access data as a Web Service.

Note: If the application server is to be deployed under COM+ (or MTS), the remote data module includes events for when the application server is activated or deactivated. This allows it to acquire database connections when activated and release them when deactivated.

The contents of the remote data module

As with any data module, you can include any nonvisual component in the remote data module. There are certain components, however, that you must include:

It must include a dataset component to represent the records from that database server if the remote data module is exposing information from a database server. Other components, such as a database connection component of some type, may be required to allow the dataset to interact with a database server.

For every dataset that the remote data module exposes to clients, it must include a dataset provider. A dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server.

It must include an XML provider for every XML document that the remote data module exposes to clients. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server.

Note: Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The connection components in multi-tiered applications can be found on the DataSnap category or WebServices category of the **Tool palette**.

Using Transactional Data Modules

You can write an application server that takes advantage of special services for distributed applications that are supplied by COM+ (under Windows 2000 and later) or MTS (before Windows 2000). To do so, create a transactional data module instead of an ordinary remote data module.

When you use a transactional data module, your application can take advantage of the following special services:

- **Security.** COM+ (or MTS) provides role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module's interface. The MTS data module implements the IsCallerInRole method, which you lets you check the role of the currently connected client and conditionally allow certain functions based on that role. .
- **Database handle pooling.** Transactional data modules automatically pool database connections that are made via ADO or (if you are using MTS and turn on MTS POOLING) the BDE. When one client is finished with a database connection, another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database connection component should set its *KeepConnection* property to *False*, so that your application maximizes the sharing of connections. .
- **Transactions**. When using a transactional data module, you can provide enhanced transaction support beyond that available with a single database connection. Transactional data modules can participate in transactions that span multiple databases, or include functions that do not involve databases at all. For more information about the transaction support provided by transactional objects such as transactional data modules, see Managing transactions in multi-tiered applications.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up resources such as database handles when they are not in use.

Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can't use the database connection while it is associated with another client's remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see Supporting state information in remote data modules.

By default, all automatically generated calls to a transactional data module are transactional (that is, they assume that when the call exits, the data module can be deactivated and any current transactions committed or rolled back). You can write a transactional data module that depends on persistent state information by setting the AutoComplete property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation unless you use a custom interface.

Warning: Application servers containing transactional data modules should not open database connections until the data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, add code to open database connections when the data module is activated and close them when it is deactivated.

Pooling Remote Data Modules

Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol.

If you are creating a transactional data module that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object.

Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using *TWebConnection*. Under this second type of object pooling, you limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the Web Server application (which passes calls to your remote data module) receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

To set up object pooling when using a Web connection (HTTP), your remote data module must override the UpdateRegistry method. In the overridden method, call RegisterPooled when the remote data module registers and UnregisterPooled when the remote data module unregisters.

When using either method of object pooling, your remote data module must be stateless. This is because a single instance potentially handles requests from several clients. If it relied on persistent state information, clients could interfere with each other. See Supporting State Information in Remote Data Modules for more information on how to ensure that your remote data module is stateless.

Choosing a Connection Protocol

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans.

The following topics describe the unique features for each connection protocol:

- Using DCOM Connections
- Using Socket Connections
- Using Web Connections
- Using SOAP Connections

Using DCOM Connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server.

DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (COM+ or MTS). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. Because of this, it is impossible to assign roles to separate clients: The runtime executable is, effectively, the only client.

Using Socket Connections

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application, you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Working with Sockets.

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

Note: ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to EnableSocketTransport in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

Note: Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the **Connections** ▸ **Registered Objects Only** menu item on ScktSrvr.exe.

When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can't release its resources because it is unaware that the client has gone away.

Using Web Connections

HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Creating Internet Server Applications

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of Internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to EnableWebTransport in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling. This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed.

Unlike most other connection components, you can't use callbacks when the connection is formed via HTTP.

Using SOAP Connections

SOAP is the protocol that underlies the built-in support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a "firewall." For more information about using SOAP to distribute applications, see Using Web Services..

As with HTTP connections, you can't use callbacks when the connection is formed via SOAP.

Building a Multi-tiered Application

To create a multi-tiered database application

1. Create the application server.
2. Register or install the application server.
3. Create a client application.

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the **Object Inspector**.

Note: If you are not creating the client application on the same system as the server, and you are using a DCOM connection, you may want to register the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the **Object Inspector**. (If you are using a Web connection, SOAP connection, or socket connection, the connection component fetches the names of registered providers from the server machine.)

Creating the Application Server

You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.

To create an application server

1 Start a new project:

- If you are using SOAP as a transport protocol, this should be a new Web Service application. Choose **File** ▸ **New** ▸ **Other**, and on the WebServices page of the new items dialog, choose SOAP Server application. Select the type of Web Server you want to use, and when prompted whether you want to define a new interface for the SOAP module, say no.
- For any other transport protocol, you need only choose **File** ▸ **New** ▸ **Application**.

Save the new project.

2 Add a new remote data module to the project. From the main menu, choose **File** ▸ **New** ▸ **Other**, and on the ActiveX, Delphi Files, or WebServices page of the new items dialog, select

- **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, or sockets.
- **Transactional Data Module** if you are creating a remote data module that runs under COM+ (or MTS). Connections can be formed using DCOM, HTTP, or sockets. However, only DCOM supports the security services.
- **SOAP Server Data Module** if you are creating a SOAP server in a Web Service application.

For more detailed information about setting up a remote data module, see Setting up the remote data module.

> **Note:** Remote data modules are more than simple data modules. The SOAP data module implements an invokable interface in a Web Service application. Other data modules are COM Automation objects.

3 Place the appropriate dataset components on the data module and set them up to access the database server.

4 Place a *TDataSetProvider* component on the data module for each dataset you want to expose to clients. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider to the name of the dataset to access. You can set additional properties for the provider. See Using provider components for more detailed information about setting up a provider.

If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.

5 Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to

- Extend the application server's interface to provide additional ways for the client application to call the server.
- Provide transaction support beyond the transactions automatically created when applying updates.
- Create master/detail relationships between the datasets in your application server.
- Ensure your application server is stateless.
- Divide your application server into multiple remote data modules.

6 Save, compile, and register or install the application server.

7 If your server application does not use DCOM or SOAP, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.

- For TCP/IP sockets this is a socket dispatcher application, Scktsrvr.exe.
- For HTTP connections this is httpsrvr.dll, an ISAPI/NSAPI DLL that must be installed with your Web server.

Setting Up the Remote Data Module

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See The Structure of the Application Server for information on what type of remote data module you need.

The following topics describe how to configure each type of remote data module:

- Configuring TRemoteDataModule
- Configuring TMTSDataModule
- Configuring TSoapDataModule

Configuring TRemoteDataModule

To add a *TRemoteDataModule* component to your application, choose **File** ▸ **New** ▸ **Other** and select Remote Data Module from the ActiveX page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to the new interface. For more information, see Extending the application server's interface.

You must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its *AutoSessionName* property set to *True* to handle threading issues on BDE-enabled datasets).
- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.
- If you choose Neutral, the remote data module can receive simultaneous calls on separate threads, as in the Free-threaded model, but COM guarantees that no two threads access the same method at the same time.

If you are creating an EXE, you must also specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

Configuring TMTSDataModule

To add a *TMTSDataModule* component to your application, choose **File** ▸ **New** ▸ **Other** and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to your new interface. For more information, see Extending the application server's interface.

You must specify the threading model in the Transactional Data Module wizard. Choose Single, Apartment, or Both.

- If you choose Single, client requests are serialized so that your application services only one at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment, the system ensures that any instance of your remote data module services one request at a time, and calls always use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager.
- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

Note: The Apartment model under MTS or COM+ is different from the corresponding model under DCOM.

You must also specify the transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of a transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of a transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of transactions.

Configuring TSOAPDataModule

To add a *TSoapDataModule* component to your application, choose **File** ▸ **New** ▸ **Other** and select SOAP Server Data Module from the WebServices page of the new items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendantthat your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TSoapDataModule*, which implements *IMyDataServer*, a descendant of *IAppServerSOAP*.

Note: To use *TSoapDataModule*, the new data module should be added to a Web Service application. The *IAppServerSOAP* interface is an invokable interface, which is registered in the initialization section of the new unit. This allows the invoker component in the main Web module to forward all incoming calls to your data module.

You may want to edit the definitions of the generated interface and *TSoapDataModule* descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name or GUID can use any of the properties and methods that you add.

Extending the Interface of the Application Server

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module.

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. See Working with type libraries for more information about using the type library editor .

Note: Neither of these approaches works if you are implementing *TSoapDataModule*. For *TSoapDataModule* descendants, you must edit the server interface directly.

When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).

Note: You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

Once you have added to your remote data module's interface, locate the properties and methods that were added to your remote data module's implementation. Add code to finish this implementation by filling in the bodies of the new methods.

If you are not writing a SOAP data module, client applications call your interface extensions using the *AppServer* property of their connection component. With SOAP data modules, they call the connection component's *GetSOAPServer* method. For more information on how to call your interface extensions, see Calling server interfaces.

Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed. However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP or SOAP-based connection. TWebConnection and TSoapConnection do not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the SupportCallbacks property. All other types of connection automatically support callbacks.

Extending a transactional application server's interface

When using transactions or just-in-time activation, you must be sure all new methods call SetComplete to indicate when they are finished. This allows transactions to complete and permits the remote data module to be deactivated.

Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server unless they provide a safe reference. If you are using a stateless MTS

data module, neglecting to use a safe reference can lead to crashes because you can't guarantee that the remote data module is active.

Managing Transactions in Multi-tiered Applications

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see Managing transactions.

If you have a transactional data module, you can broaden your transaction support by using COM+ (or MTS) transactions. These transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, they can span multiple databases.

Only the BDE- and ADO-based data access components support two-phase commit. Do not use InterbaseExpress or dbExpress components if you want to have transactions that span multiple databases.

Warning: When using the BDE, two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

By default, all *IAppServer* calls on a transactional data module are transactional. You need only set the transaction attribute of your data module to indicate that it must participate in transactions. In addition, you can extend the application server's interface to include method calls that encapsulate transactions that you define.

If your transaction attribute indicates that the remote data module requires a transaction, then every time a client calls a method on its interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

Note: Do not combine COM+ or MTS transactions with explicit transactions created by a database connection component or using explicit SQL commands. When your transactional data module is enlisted in a transaction, it automatically enlists all of your database calls in the transaction as well.

Supporting Master/detail Relationships

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see Creating Master/detail Relationships.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see Limiting records with parameters.)
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

Supporting State Information in Remote Data Modules

The *IAppServer* interface, which client datasets use to communicate with providers on the application server, is mostly stateless. When an application is stateless, it does not "remember" anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation or object pooling. SOAP data modules must be stateless.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must "remember" information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (*AS_ApplyUpdates*, *AS_Execute*, *AS_GetParams*, *AS_GetRecords*, or *AS_RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

When the provider packages a set of records in a data packet, it notes the value of CUST_NO on the last record in the packet:

```
[Delphi]
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
end;
```

```
[C++]
TRemoteDataModule1::DataSetProvider1GetData(TObject *Sender, TCustomClientDataSet *DataSet)
{
  DataSet->Last(); // move to the last record
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  pProvider->Tag = DataSet->FieldValues["CUST_NO"];
}
```

The provider sends this last CUST_NO value to the client after sending the data packet:

```
[Delphi]
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
end;
```

```
[C++]
TRemoteDataModule1::DataSetProvider1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  OwnerData = pProvider->Tag;
}
```

On the client, the client dataset saves this last value of CUST_NO:

```
[Delphi]
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
end;
```

```
[C++]
TDataModule1::ClientDataSet1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pDS = dynamic_cast<TComponent *>(Sender);
  pDS->Tag = OwnerData;
}
```

Before fetching a data packet, the client sends the last value of CUST_NO it received:

```
[Delphi]
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { Send last value of CUST_NO to application server }
  end;
end;
```

```
[C++]
TDataModule1::ClientDataSet1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TClientDataSet *pDS = dynamic_cast<TClientDataSet *>(Sender);
  if (!pDS->Active)
    return;
  OwnerData = pDS->Tag;
}
```

Finally, on the server, the provider uses the last CUST_NO sent as a minimum value in the query:

```
[Delphi]
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
                    var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { force the query to reexecute }
    end;
end;
```

```
[C++]
TRemoteDataModule1::DataSetProvider1BeforeGetRecords(TObject *Sender, OleVariant
&OwnerData)
{
  if (!VarIsEmpty(OwnerData))
  {
    TDataSetProvider *pProv = dynamic_cast<TDataSetProvider *>(Sender);
    TSQLDataSet *pDS = (dynamic_cast<TSQLDataSet *>(pProv->DataSet);
    pDS->Params->ParamValues["MinVal"] = OwnerData;
    pDS->Refresh(); // force the query to reexecute
  }
}
```

Using Multiple Remote Data Modules

You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple units, where each unit is relatively self-contained.

Although you can always create multiple remote data modules on the application server that function independently, a special connection component on the DataSnap category of the **Tool palette** provides support for a model where you have one main "parent" remote data module that dispatches connections from clients to other "child" remote data modules. This model requires that you use a COM-based application server (that is, not *TSoapDataModule*).

To create the parent remote data module, you must extend its *IAppServer* interface, adding properties that expose the interfaces of the child remote data modules. That is, for each child remote data module, add a property to the parent data module's interface whose value is the *IAppServer* interface for the child data module. The property getter should look something like the following:

```
function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  if not Assigned(ChildRDMFactory) then
    ChildRDMFactory :=
      TComponentFactory.Create(ComServer, TChildRDM, Class_ChildRDM,
                               ciInternal, tmApartment);
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.MainRDM := Self;
end;
```

For information about extending the parent remote data module's interface, see Extending the application server's interface.

Tip: You may also want to extend the interface for each child data module, exposing the parent data module's interface, or the interfaces of the other child data modules. This lets the various data modules in your application server communicate more freely with each other.

Once you have added properties that represent the child remote data modules to the main remote data module, client applications do not need to form separate connections to each remote data module on the application server. Instead, they share a single connection to the parent remote data module, which then dispatches messages to the "child" data modules. Because each client application uses the same connection for every remote data module, the remote data modules can share a single database connection, conserving resources. For information on how child applications share a single connection, see Connecting to an Application Server That Uses Multiple Data Modules.

Registering the Application Server

Before client applications can locate and use an application server, it must be registered or installed.

- If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see Registering a COM Object.
- If you are using a transactional data module, you do not register the application server. Instead, you install it with COM+ or MTS. .
- When the application server uses SOAP, the application must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, you need to publish a WSDL document that describes the invokable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see Generating WSDL Documents for a Web Service Application.

Creating the Client Application

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application

1. Add a new data module to the project.
2. Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See The Structure of the Client Application for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the Application Server.
4. Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see Managing Server Connections.
5. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Using Client Datasets.
6. Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.
7. Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:

- Your application may want to make direct calls to the application server. Calling Server Interfaces describes how to do this.
- You may want to use the special features of client datasets that support their interaction with the provider components. These are described in Using a Client Dataset with a Provider.

Connecting to the Application Server

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap or WebServices category of the Tool Palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See Choosing a connection protocol for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol. See the following topics for details:
- Specifying a connection using DCOM
- Specifying a connection using sockets
- Specifying a connection using HTTP
- Specifying a connection using SOAP
- Identify the application server on the server machine.
- If you are not using SOAP, identify the server using the ServerName or ServerGUID property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See Setting up the remote data module for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the **Object Inspector**. *ServerGUID* specifies the GUID of the remote data module's interface. You can look up this value using the type library editor.
- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

If you are using SOAP, the server is identified in the URL you use to locate the server machine. Follow the steps in Specifying a connection using SOAP.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

Specifying a Connection Using DCOM

When using DCOM to communicate with the application server, client applications include a TDCOMConnection component for connecting to the application server. *TDCOMConnection* uses the ComputerName property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

Note: Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for *ComputerName*. For more information, see Brokering connections.

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

Specifying a Connection Using Sockets

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a TSocketConnection component for connecting to the application server.

TSocketConnection identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see Describing sockets.

Three properties of *TSocketConnection* specify this information:

- Address specifies the IP Address of the server.
- Host specifies the host name of the server.
- Port specifies the port number of the socket dispatcher program on the application server.

Address and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see Describing the host.

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for *Address* or *Host*. For more information, see Brokering connections.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher program that forwards incoming messages to your application server. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

Note: You can configure the port of the socket dispatcher while it is running by right-clicking the Borland Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption.

To add your own encryption

1. Create a COM object that supports the IDataIntercept interface. This is an interface for encrypting and decrypting data.
2. Use TPacketInterceptFactory as the class factory for this object. If you are using a wizard to create the COM object in step 1, replace the line in the initialization section that says `TComponentFactory.Create(...)` with `TPacketInterceptFactory.Create(...)`.
3. Register your new COM server on the client machine.
4. Set the InterceptName or InterceptGUID property of the socket connection component to specify this COM object. If you used *TPacketInterceptFactory* in step 2, your COM server appears in the drop-down list of the **Object Inspector** for the *InterceptName* property.
5. Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept Name or Intercept GUID to the ProgId or GUID for the interceptor.

This mechanism can also be used for data compression and decompression.

Specifying a Connection Using HTTP

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a TWebConnection component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server and httpsrvr.dll, and the path to the Web server application (httpsrvr.dll). Specify this value using the URL property.

Note: When using *TWebConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the UserName and Password properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for URL. For more information, see Brokering connections.

Specifying a Connection Using SOAP

You can establish a connection to a SOAP application server using the TSoapConnection component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security and to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web Service provider that implements the *IAppServerSOAP* or *IAppServer* interface. (The UseSOAPAdapter property specifies which interface it expects the server to support.) If the server implements the *IAppServerSOAP* interface, *TSoapConnection* converts that interface to an *IAppServer* interface for client datasets. *TSoapConnection* locates the Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the URL property.

By default, *TSOAPConnection* automatically looks for an *IAppServerSOAP* (or *IAppServer*) interface. If the server includes more than one remote data module, you must indicate the target data module's interface (an *IAppServerSOAP* descendant) so that *TSOAPConnection* can identify the remote data module you want to use. There are two ways to do this:

- Set the SOAPServerIID property to indicate the interface of the target remote data module. This method works for any server that implements an *IAppServerSOAP* descendant. *SOAPServerIID* identifies the target interface by its GUID. At runtime, you can use the interface name, and the compiler automatically extracts the GUID. However, at design time, in the **Object Inspector**, you must specify the GUID string.
- If the server is written using the Delphi language, you can simply include the name of the SOAP data module's interface following a slash at the end of the path portion of the URL. This lets you specify the interface by name rather than GUID, but is only available when both client and server are written in Delphi.

Tip: The first approach, using the *SOAPServerIID* method, has the added advantage that it lets you call extensions to the remote data module's interface.

If you are using a proxy server, you must indicate the name of the proxy server using the Proxy property. If that proxy requires authentication, you must also set the values of the UserName and Password properties so that the connection component can log on.

Note: When using *TSoapConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

Brokering Connections

If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host, RemoteHost*, or *URL*). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the ObjectBroker property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL* to disk.

Note: You can not use the *ObjectBroker* property with SOAP connections.

Managing Server Connections

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

The following topics describe how to use a connection component for

- Connecting to the Server.
- Dropping or Changing a Server Connection.
- Calling Server Interfaces.
- Connecting to an Application Server that Uses Multiple Data Modules.

Connecting to the Server

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in Connecting to the application server. Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

Dropping or Changing a Server Connection

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

Note: Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

Calling Server Interfaces

Applications do not need to call the *IAppServer* or *IAppServerSOAP* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* or *IAppServerSOAP* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface, you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the AppServer property of the connection component.

AppServer is a Variant that represents the application server's interface. If you are not using SOAP, you can call an interface method using *AppServer* by writing a statement such as

```
[Delphi]
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

Using early binding with DCOM

When you are using DCOM as a communications protocol, you can use early binding of *AppServer* calls. Use the as operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module. For example:

```
[Delphi]
with MyConnection.AppServer as IMyAppServer do
SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use TRegsvr.exe, which ships with Delphi to register the type library.

Note: See the TRegSvr demo (which provides the source for TRegsvr.exe) for an example of how to register the type library programmatically.

Using dispatch interfaces with TCP/IP or HTTP

When you are using TCP/IP or HTTP, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp' appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
[Delphi]
var
TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
...
TempInterface.SpecialMethod(x,y);
...
end;
```

Note: To use the dispinterface, you must add the _TLB unit that is generated when you save the type library to the **uses** clause of your client module.

Calling the interface of a SOAP-based server

If you are using SOAP, you can't use the *AppServer* property. Instead, you must obtain the server's interface by calling the *GetSOAPServer* method. Before you call *GetSOAPServer*, however, you must take the following steps:

- Your client application must include the definition of the application server's interface and register it with the invocation registry. You can add the definition of this interface to your client application by referencing a WSDL document that describes the interface you want to call. For information on importing a WSDL document that describes the server interface, see Importing WSDL documents. When you import the interface definition, the WSDL importer automatically adds code to register it with the invocation registry. For more information about interfaces and the invocation registry, see Understanding invokable interfaces.
- The *TSOAPConnection* component must have its UseSOAPAdapter property set to *True*. This means that the server must support the *IAppServerSOAP* interface. If the application server is built using Delphi 6 or Kylix 1, it does not support *IAppServerSOAP* and you must use a separate *THTTPRio* component instead. For details on how to call an interface using a *THTTPRio* component, see Calling invokable interfaces.
- You must set the SOAPServerIID property of the SOAP connection component to the GUID of the server interface. You must set this property before your application connects to the server, because it tells the *TSOAPConnection* component what interface to fetch from the server.

Assuming the previous three conditions are met, you can fetch the server interface as follows:

```
[Delphi]
with MyConnection.GetSOAPServer as IMyAppServer do
SpecialMethod(x,y);
```

```
[C++]
IDispatch* disp = (IDispatch*)(MyConnection->AppServer)
IMyAppServerDisp TempInterface( (IMyAppServer*)disp);
TempInterface.SpecialMethod(x,y);
```

Connecting to an Application Server That Uses Multiple Data Modules

If a COM-based application server uses a main "parent" remote data module and several child remote data modules, as described in Using multiple remote data modules, then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module.

While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single connection to the application server that is shared by all the connection components. That is, you add a single connection component that connects to the "main" remote data module on the application server, and then, for each "child" remote data module, add an additional component that shares the connection to the main remote data module.

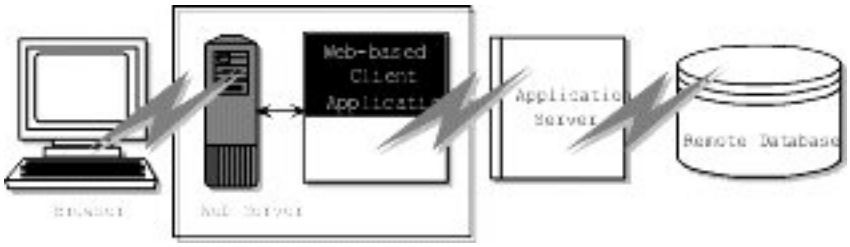
To use a single shared connection

1. For the connection to the main remote data module, add and set up a connection component as described in Connecting to the Application Server. The only limitation is that you can't use a SOAP connection.
2. For each child remote data module, use a TSharedConnection component.
   - Set its *ParentConnection* property to the connection component you added in step 1. The *TSharedConnection* component shares the connection that this main connection establishes.
   - Set its *ChildName* property to the name of the property on the main remote data module's interface that exposes the interface of the desired child remote data module.

When you assign the *TSharedConnection* component placed in step 2 as the value of a client dataset's *RemoteServer* property, it works as if you were using an entirely independent connection to the child remote data module. However, the *TSharedConnection* component uses the connection established by the component you placed in step 1.

Writing Web-based Client Applications

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in the following figure.



There are two approaches that you can take to build the Web application:

- You can combine the multi-tiered database architecture with an ActiveX form to distribute the client application as an ActiveX control. This allows any browser that supports ActiveX to run your client application as an in-process server.
- You can use XML data packets to build an InternetExpress application. This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an HTML-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

Warning: Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

Distributing a Client Application as an ActiveX Control

The multi-tiered database architecture can be combined with ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server as you would for any other multi-tiered application.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See Creating an Active Form for the Client Application for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must have installed DCOM95, which is available from Microsoft.

Creating an Active Form for the Client Application

To create an Active Form for the Client Application

1. Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in "Working with Sockets."
2. Create the client application following the steps described in Creating the client application except start by choosing **File** ▸ **New** ▸ **ActiveX** ▸ **Active Form**, rather than beginning an ordinary client project.
3. If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
4. When your client application is finished, compile the project, and select **Project** ▸ **Web Deployment Options**. In the Web Deployment Options dialog, you must do the following:

   On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine.

   On the Additional Files page, include midas.dll with your client application.
5. Finally, select **Project** ▸ **WebDeploy** to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

Building Web Applications Using InternetExpress

A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries of database functions, and the Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. This combination of features is called InternetExpress.

Before building an InternetExpress application, you should understand the Web server application architecture and the multi-tiered database architecture. These are described in Creating Internet Server Applications and Creating multi-tiered Applications

An InternetExpress application extends the basic Web server application architecture to act as the client of an application server. InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets on the client machine.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions to its clients.

Tip: You can create an InternetExpress application to provide Web browsers with "live" data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

Building an InternetExpress Application

The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled

Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).

To build a Web application using InternetExpress

1. Choose **File** ▸ **New** ▸ **Other** to display the New Items dialog box, and on the New page select Web Server application. This process is described in Creating Web server applications with Web Broker.
2. From the DataSnap category of the **Tool palette**, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See Choosing a connection protocol for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the application server.
4. Instead of a client dataset, add an TXMLBroker from the InternetExpress category of the **Tool palette** to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through an *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
5. Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see Using an XML broker.
6. Add an InternetExpress page producer (*TInetXPageProducer*) to the Web module for each separate page that users will see in their browsers. For each page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries that augment its generated HTML controls with data management capabilities.
7. Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see Adding actions to the dispatcher.
8. Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipsis button in the **Object Inspector** next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see Creating Web pages with an InternetExpress page producer.
9. Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the script name portion of the URL and the name of the Web Page component as the pathinfo portion.

Using the Javascript Libraries

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship in the source/webmidas directory:

Javascript libraries

Library	Description
xmldom.js	This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later.
xmldb.js	This library defines data access classes that manage XML data packets and XML delta packets.
xmldisp.js	This library defines classes that associate the data access classes in xmldb with HTML controls in the HTML page.

xmlerrdisp.js	This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer.
xmlshow.js	This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging.

Once you have installed these libraries, you must set the *IncludePathURL* property of all InternetExpress page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

Granting Permission to Access and Launch the Application Server

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with EOLE_ACCESS_ERROR.

Note: Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server.

To configure your application server

1. When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
2. Click the Properties button. When the dialog changes, select the Security page.
3. Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
4. Select Use Custom Launch Permissions, and press the Edit button. Add IUSR_computername to this list as well.
5. Click the Apply button.

Using an XML Broker

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it uses the *IAppServer* interface, which it acquires from a connection component.

Note: Even when using SOAP, where the application server supports *IAppServerSOAP*, the XML broker uses *IAppServer* because the connection component acts as an adapter between the two interfaces.

You must set the following properties so that the XML producer can use the *IAppServer* interface:

- Set the RemoteServer property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the **Object Inspector**.
- Set the ProviderName property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the **Object Inspector** displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the MaxRecords property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the Params property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the RequestRecords method.

Note: When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an OnRequestRecords event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using GetXMLRecords and then edit it before supplying it to the emerging Web page.

Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the WebDispatch property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see Dispatching request messages.

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

1. When the dispatcher first passes the update message to the XML broker, it receives a BeforeDispatch event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.
2. If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an OnRequestUpdate event, where you can apply the updates yourself rather than using the default processing.
3. If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
4. If there are no update errors, the XML broker receives an OnGetResponse event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
5. If there are update errors, the XML broker receives an OnGetErrorResponse event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the ReconcileProducer to generate the content of the response message.
6. Finally, the XML broker receives an AfterDispatch event, where you can perform any final actions before sending a response back to the Web browser.

Creating Web Pages with an InternetExpress Page Producer

Each InternetExpress page producer generates an HTML document that appears in the browsers of your application's clients. If your application includes several separate Web documents, use a separate page producer for each of them.

The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component. As with other page producers, you can assign it as the Producer property of an action item or call it explicitly from an OnAction event handler. For more information about using content producers with action items, see Responding to request messages with action items.

The InternetExpress page producer has a default template as the value of its HTMLDoc property. This template contains a set of HTML-transparent tags that the InternetExpress page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries used for the embedded javascript on the page. This location is specified by setting the IncludePathURL property.

You can specify the components that generate parts of the Web page using the Web page editor. Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the **Object Inspector**.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the InternetExpress page producer's default template. These components become the value of the WebPageItems property. After adding the components in the order you want them, you can customize the template to add your own HTML or change the default tags.

Using the Web Page Editor

The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a InternetExpress page producer component.

Note: You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties using the **Object Inspector**.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The InternetExpress page producer can contain one of two types of item, each of which generates an HTML form:

TDataForm, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

Items you add to *TDataForm* display data in a multi-record grid (TDataGrid) or in a set of controls each of which represents a single field from a single record (TFieldGroup). In addition, you can add a set of buttons to navigate through data or post updates (TDataNavigator), or a button to apply updates back to the Web client (TApplyUpdatesButton). Each of these items contains subitems to represent individual fields or buttons. Finally, as with most Web items, you can add a layout grid (TLayoutGroup), that lets you customize the layout of any items it contains.

TQueryForm, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

Items you add to *TQueryForm* display application-defined values(TQueryFieldGroup) or a set of buttons to submit or reset those values (TQueryButtons). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (Internet Explorer).

Setting Web Item Properties

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the XMLBroker property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the XMLDataSetField property. If the Web item represents a specific field or parameter value, the Web item has a FieldName or ParamName property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the Style property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as,

```
color: red.
```

You can define a style sheet that defines a set of style definitions. Each definition includes a style selector (the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces,

```
H2 B  {color: red}
.MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
```

The entire set of definitions is maintained by the InternetExpress page producer as its Styles property. Each Web item can then reference the styles with user-defined names by setting its StyleRule property.

If you are sharing a style sheet with other applications, you can also supply the style definitions as the value of the InternetExpress page producer's StylesFile property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

Customizing the InternetExpress Page Producer Template

The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the HTMLDoc property. This default template has the form

```
<HTML>
```

```
<HEAD>
```

```
</HEAD>
```

```
<BODY>
```

```
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
```

```
</BODY>
```

```
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

<#INCLUDES> generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
```

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
```

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </
SCRIPT>
```

<#STYLES> generates the statements that defines a style sheet from definitions listed in the Styles or StylesFile property of the InternetExpress page producer.

<#WARNINGS> generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

<#FORMS> generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is generated in the order it appears in *WebPageItems*.

<#SCRIPT> generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the HTMLFile property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The InternetExpress page producer automatically translates these tags when you call the Content method. In addition, The InternetExpress page producer automatically translates three additional tags:

<#BODYELEMENTS> is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

<#COMPONENT Name=WebComponentName> is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the IWebContent interface and has the same Owner as the InternetExpress page producer.

<#DATAPACKET XMLBroker=BrokerName> is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the InternetExpress page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the InternetExpress page producer encounters a tag that is not one of the seven types it translates automatically, it generates an OnHTMLTag event, where you can write code to perform your own translations. For more information about HTML templates in general, see HTML templates.

Tip: The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same, no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a <#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

Using XML in database applications

Using XML in Database Applications

In addition to the support for connecting to database servers, Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Working with XML Documents.

Support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. To use these components, you must first define the transformation between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets.
- provide data from and resolve updates to an XML document.
- use an XML document as the client of a provider.

Defining Transformations

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document. A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility that ships in the bin directory.

Mapping Between XML Nodes and Data Packet Fields

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone="yes" ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the XML component support in Delphi.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule. Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

One natural mapping between this document and a dataset would map each e-mail message to a single record. The record would have fields for the sender's name and address. Because an e-mail message can have multiple recipients, the recipient (<to> would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the e-mail above would map to a dataset something like the following:

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

where the nested dataset in the "To" field is

Name	Address
Joe Engineer	jengineer@MyCo.Com

the nested dataset in the "CC" field is

Name	Address
Robin Smith	rsmith@MyCo.Com

Leonard Devon	ldevon@MyCo.Com

and the nested dataset in the "Attach" field is

Attachfile
XMLSpec.txt
Schedule.txt

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...<head/> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```

Using XMLMapper

The XML mapper utility, xmlmapper.exe, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

Note: XML mapper relies on two .DLLs (midas.dll and msxml.dll) to work correctly. Be sure that you have both of these .DLLs installed before you try to use xmlmapper.exe. In addition, msxml.dll must be registered as a COM server. You can register it using Regsvr32.exe.

Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing **File** ▸ **Open** and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing **File** ▸ **Open** and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's *SaveToFile* method.) If you have not saved the data packet to disk, you can fetch the data packet directly from the application server of a multi-tiered application by right-clicking in the Datapacket pane and choosing Connect To Remote Server.

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the Mapping page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document, double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes. Once you have specified the way nodes are to be mapped, choose **Create** ▸ **Datapacket from XML**. The corresponding data packet is automatically generated and displayed in the Datapacket pane.
- If you are generating an XML document from a data packet, choose **Create** ▸ **XML from Datapacket**. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, the way you name them indicates whether they map to a tagged element with a value or to an attribute. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

Generating transformation files

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

To generate a transformation file

1. First select the radio button that indicates what the transformation creates:

- Choose the Datapacket to XML button if the mapping goes from data packet to XML document.

- Choose the XML to Datapacket button if the mapping goes from XML document to data packet.

2. If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.
3. Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.
4. Finally, choose **File** ▸ **Save** ▸ **Transformation** to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

Converting XML Documents into Data Packets

Once you have created a transformation file that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application.

The TXMLTransform component transforms an XML document into a data packet according to the mapping in a transformation file.

Note: You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

Specifying the source XML document

There are three ways to specify the source XML document:

- If the source document is an .xml file on disk, you can use the SourceXmlFile property.
- If the source document is an in-memory string of XML, you can use the SourceXml property.
- If you have an IDOMDocument interface for the source document, you can use the SourceXmlDocument property.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

Specifying the transformation

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the TransformationFile property to indicate a transformation file that was created using xmlmapper.exe.
- Set the TransformationDocument property if you have an *IDOMDocument* interface for the transformation.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

Obtaining the resulting data packet

To cause *TXMLTransform* to perform its transformation and generate a data packet, you need only read the Data property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to a client dataset:

```
[Delphi]
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

```
[C++]
XMLTransform1->SourceXMLFile = "CustomerDocument.xml";
XMLTransform1->TransformationFile = "CustXMLToCustTable.xtr";
ClientDataSet1->XMLData = XMLTransform1->Data;
```

Converting user-defined nodes

When you define a transformation using xmlmapper.exe, you can specify that some of the nodes in the XML document are "user-defined." User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the OnTranslate event. The *OnTranslate* event handler is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the OnTranslate event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

into a single field value:

```
[Delphi]
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
  CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
```

```
    end;
  end;
end;
```

```
[C++]
void __fastcall TForm1::XMLTransform1Translate(TObject *Sender, AnsiString Id,
_di_IDOMNode SrcNode, AnsiString &Value, _di_IDOMNode DestNode)
{
  if (Id == "FullName")
  {
    Value = "";
    if (SrcNode.hasChildNodes)
    {
      _di_IXMLDOMNode CurNode = SrcNode.firstChild;
      Value = SrcValue + AnsiString(CurNode.nodeValue);
      while (CurNode != SrcNode.lastChild)
      {
        CurNode = CurNode.nextSibling;
        Value = Value + AnsiString(" ");
        Value = Value + AnsiString(CurNode.nodeValue);
      }
    }
  }
}
```

Using an XML Document as the Source for a Provider

The TXMLTransformProvider component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Using Provider Components. For information on using provider components with client datasets, see Using a Client Dataset with a Provider.

You can specify the XML document from which the XML provider provides data and to which it applies updates using the XMLDataFile property.

TXMLTransformProvider components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the TransformRead and TransformWrite properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component's TransformationFile or TransformationDocument property, just as when using a stand-alone TXMLTransform component.

In addition, if the transformation includes any user-defined nodes, you must supply an OnTranslate event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider's *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using *TransformRead.XmlSource* or *TransformRead.XmlSourceDocument*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

Using an XML Document as the Client of a Provider

The TXMLTransformClient component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransformClient* lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the ProviderName property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Using Provider Components.

If the provider is on a remote application server, you must use a DataSnap connection component to connect to that application server. Specify the connection component using the RemoteServer property. For information on DataSnap connection components, see Connecting to the Application Server.

Fetching an XML document from a provider

TXMLTransformClient uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this *TXMLTransform* component as the value of the TransformGetData property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component's TransformationFile or TransformationDocument property, just as when using a stand-alone TXMLTransform component. If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an OnTranslate event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the SetParams method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

Note: You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on TransformSetParams property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the properties you can use, see Converting XML Documents Into Data Packets.

Once you have configured *TransformGetData* and supplied any input parameters, you can call the GetDataAsXml method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

```
[Delphi]
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
```

```
  finally
    XMLDoc.Free;
  end;
end;
```

```
[C++]
XMLTransformClient1->ProviderName = "Provider1";
XMLTransformClient1->TransformGetData->TransformationFile = "CustTableToCustXML.xtr";
XMLTransformClient1->TransFormSetParams->SourceXmlFile = "InputParams.xml";
XMLTransformClient1->SetParams("", "InputParamsToDP.xtr");
AnsiString XML = XMLTransformClient1->GetDataAsXml();
TFileStream pXMLDoc = new TFileStream("Customers.xml", fmCreate || fmOpenWrite);
__try
{
  pXMLDoc->Write(XML.c_str(), XML.Length());
}
__finally
{
  delete pXMLDoc;
}
```

Applying updates from an XML document to a provider

TXMLTransformClient also lets you insert all of the data from an XML document into the provider's dataset or to delete all of the records in an XML document from the provider's dataset. To perform these updates, call the ApplyUpdatesmethod,passing in

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can't be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can't be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document Customers.xml into a delta packet and applies all updates regardless of the number of errors:

```
[Delphi]
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```

```
[C++]
StringList1->LoadFromFile("Customers.xml");
nErrors = ApplyUpdates(StringList1->Text, "CustXMLToInsert.xtr", -1);
```

Writing Internet Applications

Creating Internet server applications

Creating Internet Applications: Overview

Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Many operations that you can perform with an ordinary application can be incorporated into a Web server application.

The IDE provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It provides additional components and new features like the Preview tab, which allows the content of a page to be displayed without the developer having to run the application. Applications developed with WebSnap can include Web Broker components, whereas applications developed with Web Broker cannot include WebSnap components.

About Web Broker and WebSnap

Part of the function of any application is to make data accessible to the user. In a standard application you accomplish this by creating traditional front end elements, like dialogs and scrolling windows. Developers can specify the exact layout of these objects using familiar form design tools. Web server applications must be designed differently, however. All information passed to users must be in the form of HTML pages which are transferred through HTTP. Pages are generally interpreted on the client machine by a Web browser application, which displays the pages in a form appropriate for the user's particular system in its present state.

The first step in building a Web server application is choosing which architecture you want to use, Web Broker or WebSnap. Both approaches provide many of the same features, including

- Support for CGI and Apache DSO Web server application types. These are described in Types of Web Server Applications.
- Multithreading support so that incoming client requests are handled on separate threads.
- Caching of Web modules for quicker responses.

Both the Web Broker and WebSnap components handle all of the mechanics of page transfer. WebSnap uses Web Broker as its foundation, so it incorporates all of the functionality of Web Broker's architecture. WebSnap offers a much more powerful set of tools for generating pages, however. Also, WebSnap applications allow you to use server-side scripting to help generate pages at runtime. Web Broker does not have this scripting capability. The tools offered in Web Broker are not nearly as complete as those in WebSnap, and are much less intuitive. If you are developing a new Web server application, WebSnap is probably a better choice of architecture than Web Broker.

The major differences between these two approaches are outlined in the following table:

Web Broker versus WebSnap

Web Broker	WebSnap
Backward compatible	Although WebSnap applications can use any Web Broker components that produce content, the Web modules and dispatcher that contain these are new.
Only one Web module allowed in an application.	Multiple Web modules can partition the application into units, allowing multiple developers to work on the same project with fewer conflicts.
Only one Web dispatcher allowed in the application.	Multiple, special-purpose dispatchers handle different types of requests.
Specialized components for creating content include page producers, InternetExpress components, and Web Services components.	Supports all the content producers that can appear in Web Broker applications, plus many others designed to let you quickly build complex data-driven Web pages.
No scripting support.	Support for server-side scripting allows HTML generation logic to be separated from the business logic.
No built-in support for named pages.	Named pages can be automatically retrieved by a page dispatcher and addressed from server-side scripts.
No session support.	Sessions store information about an end user that is needed for a short period of time. This can be used for such tasks as login/logout support.
Every request must be explicitly handled, using either an action item or an auto-dispatching component.	Dispatch components automatically respond to a variety of requests.
Only a few specialized components provide previews of the content they produce. Most development is not visual.	WebSnaplets you build Web pages more visually and view the results at design time. Previews are available for all components.

For more information on Web Broker, see Using Web Broker. For more information on WebSnap, see Creating Web Server Applications Using WebSnap.

Terminology and Standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

- RFC822, "Standard for the format of ARPA Internet text messages," describes the structure and content of message headers.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, "Hypertext Transfer Protocol—HTTP/1.0," describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, www.ietf.cnri.reston.va.us

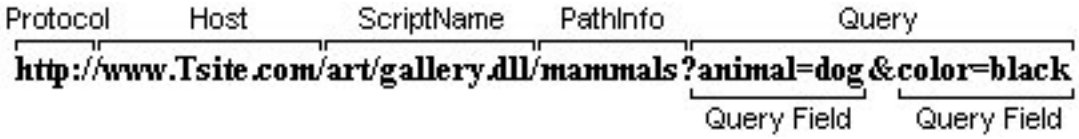
These documents include, among other information, details about

- Parts of a Uniform Resource Locator
- HTTP request header information

- HTTP server activity

Parts of a Uniform Resource Locator

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in the following figure:



The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the pathinfo. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.

URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

HTTP Request Header Information

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as "Host" followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.dll/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to

form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

HTTP Server Activity

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests:

- Composing client requests
- Serving client requests
- Responding to client requests

Composing Client Requests

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Serving Client Requests

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application:

- If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.
- If the program is a dynamic-link library (DLL), the server loads the DLL (if necessary) and passes the information contained in the request to the DLL as a structure. The server waits while the program executes. When the DLL exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

Responding to Client Requests

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent may differ based on the type of program.

When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client. Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request.

Types of Web Server Applications

Whether you use Web Broker or WebSnap, you can create five standard types of Web server applications. In addition, you can create a Web Application Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web Application Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other five types.

ISAPI and NSAPI

An ISAPI or NSAPI Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by the ISAPI/NSAPI application, which creates appropriate request and response objects. Each request message is automatically handled in a separate execution thread.

CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by the CGI application, which creates appropriate request and response objects. Each request message is handled by a separate instance of the application.

Apache

An Apache Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by the Apache Web server application, which creates appropriate request and response objects. Each request message is automatically handled in a separate execution thread. You can build your Web server applications using Apache 1 or 2 as your target type.

When you deploy your Apache Web server application, you will need to specify some application-specific information in the Apache configuration files. For example, in Apache 1 projects the default module name is the project name with _module appended to the end. For example, a project named Project1 would have Project1_module as its module name. Similarly, the default content type is the project name with -content appended, and the default handler type is the project name with-handler appended.

These definitions can be changed in the project (.dpr) file when necessary. For example, when you create your project a default module name is stored in the project file. Here is a common example:

```
[Delphi]
exports
apache_module name 'Project1_module';
```

```
[C++]
extern "C"
{
Httpd::module __declspec(dllexport) Project1_module;
}
```

Note: When you rename the project during the save process, that name isn't changed automatically. Whenever you rename your project, you must change the module name in your project file to match your project name. The content and handler definitions should change automatically once the module name is changed.

For information on using module, content, and handler definitions in your Apache configuration files, see the documentation on the Apache Web site httpd.apache.org.

Web App Debugger

The server types mentioned above have their advantages and disadvantages for production environments, but none of them is well-suited for debugging. Deploying your application and configuring the debugger can make Web server application debugging far more tedious than debugging other application types.

Fortunately, Web server application debugging doesn't need to be that complicated. The IDE includes a Web App Debugger which makes debugging simple. The Web App Debugger acts like a Web server on your development machine. If you build your Web server application as a Web App Debugger executable, deployment happens automatically during the build process. To debug your application, start it using **Run** ▸ **Run**. Next, select **Tools** ▸ **Web App Debugger**, click the default URL and select your application in the Web browser which appears. Your application will launch in the browser window, and you can use the IDE to set breakpoints and obtain debugging information.

When your application is ready to be tested or deployed in a production environment, you can convert your Web App Debugger project to one of the other target types using the steps given below.

Note: When you create a Web App Debugger project, you will need to provide a CoClass Name for your project. This is simply a name used by the Web App Debugger to refer to your application. Most developers use the application's name as the CoClass Name.

Converting Web server application target types

One powerful feature of Web Broker and WebSnap is that they offer several different target server types. The IDE allows you to easily convert from one target type to another.

Because Web Broker and WebSnap have slightly different design philosophies, you must use a different conversion method for each architecture.

To convert your Web Broker application target type

1. Right-click the Web module and choose Add To Repository.
2. In the Add To Repository dialog box, give your Web module a title, text description, Repository page (probably Data Modules), author name, and icon.
3. Choose OK to save your Web module as a template.
4. From the main menu, choose **File** ▸ **New** and select Web Server Application. In the New Web Server Application dialog box, choose the appropriate target type.
5. Delete the automatically generated Web module.

6 From the main menu, choose **File** ▸ **New** and select the template you saved in step 3. This will be on the page you specified in step 2.

To convert a WebSnap application's target type

1 Open your project in the IDE.

2 Display the **Project Manager** using **View** ▸ **Project Manager**. Expand your project so all of its units are visible.

3 In the Project Manager, click the New button to create a new Web server application project. Double-click the WebSnap Application item in the WebSnap tab. Select the appropriate options for your project, including the server type you want to use, then click OK.

4 Expand the new project in the Project Manager. Select any files appearing there and delete them.

5 One at a time, select each file in your project (except for the form file in a Web App Debugger project) and drag it to the new project. When a dialog appears asking if you want to add that file to your new project, click Yes.

Debugging Server Applications

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting.

The following topics describe techniques you can use to debug Web server applications:

- Using the Web Application Debugger
- Debugging Web Applications that are DLLs

Using the Web Application Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server.

For information on how to write this Web server application using Web Broker, see Using Web Broker. For more information on using WebSnap, see Creating Web Server applications using WebSnap.

Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it.

To launch your application with the Web Application Debugger

1 With your project loaded in the IDE, set any breakpoints so that you can debug your application just like any other executable.

2 Choose **Run** ▸ **Run**. This displays the console window of the COM server that is your Web server application. The first time you run your application, it registers your COM server so that the Web App debugger can access it.

3 Select **Tools** ▸ **Web App Debugger**.

4 Click the Start button. This displays the ServerInfo page in your default Browser.

5 The ServerInfo page provides a drop-down list of all registered Web Application Debugger executables. Select your application from the drop-down list. If you do not find your application in this drop-down list, try running your application as an executable. Your application must be run once so that it can register itself. If you still do not find your application in the drop-down list, try refreshing the Web page. (Sometimes the Web browser caches this page, preventing you from seeing the most recent changes.)

6 Once you have selected your application in the drop-down list, press the Go button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Converting your application to another type of Web server application

When you have finished debugging your Web server application with the Web Application Debugger, you will need to convert it to another type that can be installed on a commercial Web server. To learn more about converting your application, see "Converting Web server application target types" in the topic Types of Web server applications.

Debugging Web Applications That Are DLLs

ISAPI, NSAPI, and Apache applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. Because these applications are DLLs, you can debug them by setting your application's run parameters to launch the server.

To set up your application's run parameters, choose **Run** ▸ **Parameters** and set the Host Application and Run Parameters to specify the executable for the Web server and any parameters it requires when you launch it. For details about these values on your Web server, see the documentation provided by you Web server vendor.

Note: Some Web Servers require additional changes before you have the rights to launch the Host Application in this way. See your Web server vendor for details.

Tip: If you are using Windows 2000 with IIS 5, details on all of the changes you need to make to set up your rights properly are described at the following Web site:

```
http://community.borland.com/article/0,1410,23024,00.html
```

Once you have set the Host Application and Run Parameters, you can set up your breakpoints so that when the server passes a request message to your DLL, you hit one of your breakpoints, and can debug normally.

Note: Before launching the Web server using your application's run parameters, make sure that the server is not already running.

User rights necessary for DLL debugging

Under Windows, you must have the correct user rights to debug a DLL.

To obtain these rights

1 In the Administrative Tools portion of the Control Panel, click on Local Security Policy. Expand Local Policies and double-click User Rights Assignment. Double-click Act as part of the operating system in the right-hand panel.

2 Select Add to add a user to the list. Add your current user.

3 Reboot so the changes take effect.

Using Web Broker

Using Web Broker

Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

The following sections in this series explain how you use the Web Broker components to create a Web server application.

Creating Web Server Applications with Web Broker

Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

To create a new Web server application using the Web Broker architecture:

1. Select **File** ▸ **New** ▸ **Other**.
2. In the New Items dialog box, select the New tab under Delphi Projects and choose Web Server Application.
3. A dialog box appears, where you can select one of the Web server application types:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Apache: Selecting one of these two application types (1.x and 2.x) sets up your project as a DLL, with the exported methods expected by the applicable Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- Web Application Debugger stand-alone executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components.

The Web Module

The Web module (TWebModule) is a descendant of TDataModule and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as TPageProducer, TDataSetPageProducer, TDataSetTableProducer, TQueryTableProducer and TInetXPageProducer or descendants of TCustomContentProducer that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a multi-tiered database application.

In addition to storing non-visual components and business rules, the Web module also acts as a Web dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a TWebDispatcher component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from TCustomWebDispatcher, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the TWebDispatcher component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

Note: The Web module that you set up at design time is actually a template. In ISAPI and NSAPI applications, each request message spawns a separate thread, and separate instances of the Web module and its contents are created dynamically for each thread.

Warning: The Web module in a DLL-based Web server application is cached for later reuse to increase response time. The state of the dispatcher and its action list is not reinitialized between requests. Enabling or disabling action items during execution may cause unexpected results when that module is used for subsequent client requests.

The Web Application Object

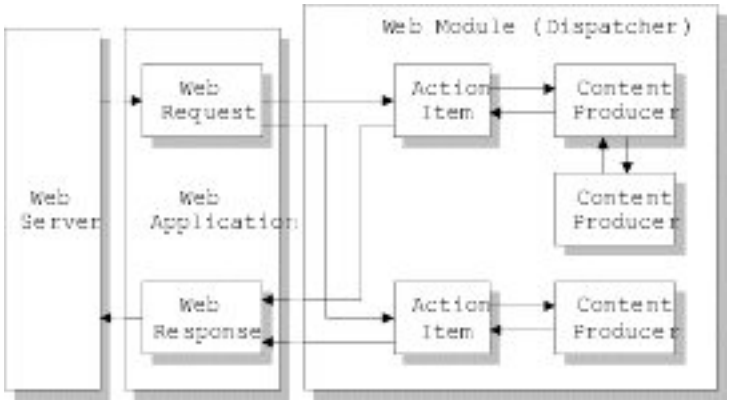
The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

Warning: Do not include the Forms or QForms unit in the project **uses** clause after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

The Structure of a Web Broker Application

When the Web application receives an HTTP request message, it creates a TWebRequest object to represent the HTTP request message, and a TWebResponse object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message.



The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message.

If you are creating the Web Client in a multi-tiered database application, your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in javascript. If you are creating a server that implements a Web Service, your Web server application may include an auto-dispatching component that passes SOAP-based messages on to an invoker that interprets and executes them. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

The Web Dispatcher

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (TWebDispatcher) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it is responsible for dispatching the request message.

Set up the Web dispatcher by adding actions to the dispatcher.

Adding Actions to the Dispatcher

Open the action editor from the **Object Inspector** by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action

that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

Dispatching Request Messages

When the dispatcher receives the client request, it generates a BeforeDispatch event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher iterates over its list of action items, looking for an entry that matches the PathInfo portion of the request message's target URL and that also provides the service specified as the method of the request message. It does this by comparing the PathInfo and MethodType properties of the *TWebRequest* object with the property of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

After checking all its action items, if the message is not handled the dispatcher checks any specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications, which are described in Building Web applications using InternetExpress

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an AfterDispatch event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

Action Items

Each action item (TWebActionItem) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Set up action items for your Web server application by

- Adding actions to the dispatcher
- Determining when action items fire
- Responding to request messages with action items

Determining When Action Items Fire

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the **Object Inspector** and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the **Object Inspector**.

The action item includes properties that specify

- The target URL
- The request method type

Other properties that influence when the dispatcher fires an action item are described in

- Enabling and disabling action items
- Choosing a default action item

The Target URL

The dispatcher compares the PathInfo property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

and assuming that the `/gallery.dll` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

The Request Method Type

The MethodType property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

MethodType values

Value	Meaning
mtGet	The request is asking for the information associated with the target URI to be returned in a response message.
mtHead	The request is asking for the header properties of a response, as if servicing an mtGet request, but omitting the content of the response.
mtPost	The request is providing information to be posted to the Web application.
mtPut	The request asks that the resource associated with the target URI be replaced by the content of the request message.
mtAny	Matches any request method type, including mtGet, mtHead, mtPut, and mtPost.

Enabling and Disabling Action Items

Each action item has an Enabled property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A BeforeDispatch event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

Warning: Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the Web server application is a DLL that caches Web modules, the initial state will not be reinitialized for the next request. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

Choosing a Default Action Item

Only one of the action items can be the default action item. The default action item is selected by setting its Default property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the Enabled property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its Enabled property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

Warning: Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be reevaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

Responding to Request Messages with Action Items

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its Producer property, that producer automatically assigns the Content of the response message using its Content method. The Internet page of the **Tool palette** includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an OnAction event. The OnAction event handler is passed the TWebRequest object that represents the HTTP request message and a TWebResponse object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see Generating the content of response messages.

Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled to False* so that others can continue, make sure the default action item leaves the Handled parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

Accessing Client Request Information

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of an object descended from *TWebRequest*. For example, in NSAPI and ISAPI applications, the request message is encapsulated by a TISAPIRequest object, and console CGI applications use TCGIRequest objects.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request, including

- Request header information
- The content of the request message

Properties That Contain Request Header Information

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the GetFieldByName method.

The request header properties can be categorized by function:

- Properties that identify the target
- Properties that describe the Web client
- Properties that identify the purpose of the request
- Properties that describe the expected response
- Properties that describe the content

Properties That Identify the Target

The full target of the request message is given by the URL property. Usually, this is a URL that can be broken down into the protocol (HTTP), Host (server system), ScriptName (server application), PathInfo (location on the host), and Query.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the QueryFields property.

Properties That Describe the Web Client

The request includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the From property), to the URI where the message originated (the Referer or RemoteHost property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the DerivedFrom property. You can also determine the IP address of the client (the RemoteAddr property), and the name and version of the application that sent the request (the UserAgent property).

Properties That Identify the Purpose of the Request

The Method property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Predefined tag names

Value	What the message requests
OPTIONS	Information about available communication options.
GET	Information identified by the URL property.
HEAD	Header information from an equivalent GET message, without the content of the response.
POST	The server application to post the data included in the Content property, as appropriate.
PUT	The server application to replace the resource indicated by the URL property with the data included in the Content property.
DELETE	The server application to delete or hide the resource identified by the URL property.
TRACE	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The MethodType property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

Properties That Describe the Expected Response

The Accept property indicates the media types the Web client will accept as the content of the response message. The IfModifiedSince property specifies whether the client only wants information that has changed recently. The Cookie property includes state information (usually added previously by your application) that can modify the response.

Properties That Describe the Content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the ContentType property, and its length in the ContentLength property. If the content of the message was encoded (for example, for data compression), this information is in the ContentEncoding property. The name and version number of the application that produced the content is specified by the ContentVersion property. The Title property may also provide information about the content.

The Content of HTTP Request Messages

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database accessed by the Web server application.

The unprocessed value of the content is given by the Content property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the ContentFields property.

Creating HTTP Response Messages

When the Web server application creates a TWebRequest descended object for an incoming HTTP request message, it also creates a corresponding object descended from *TWebResponse* to represent the response message that will be sent in return. For example, in NSAPI and ISAPI applications, the response message is encapsulated by a TISAPIResponse object, and Console CGI applications use TCGIResponse objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Responding to HTTP requests involves

- Filling in the response header
- Setting the response content
- Sending the response

Filling in the Response Header

Most of the properties of the TWebResponse object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Use the response object properties for

- Indicating the response status
- Indicating the need for client action
- Describing the server application
- Describing the content

Indicating the Response Status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the StatusCode property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the ReasonString property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

Indicating the Need for Client Action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the Location property. If the client must provide a password before you can proceed, set the WWWAuthenticate property.

Describing the Server Application

Some of the response header properties describe the capabilities of the Web server application. The Allow property indicates the methods to which the application can respond. The Server property gives the name and version number of the application used to generate the response. The Cookies property can hold state information about the client's use of the server application which is included in subsequent request messages.

Describing the Content

Several properties describe the content of the response. ContentType gives the media type of the response, and ContentVersion is the version number for that media type. ContentLength gives the length of the response. If the content is encoded (such as for data compression), indicate this with the ContentEncoding property. If the content came from another URI, this should be indicated in the DerivedFrom property. If the value of the content is time-sensitive, the LastModified property and the Expires property indicate whether the value is still valid. The Title property can provide descriptive information about the content.

Setting the Response Content

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the Content property or the ContentStream property.

The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a TFileStream object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

Note: If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

Sending the Response

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an OnAction event handler. The response object provides two methods for sending a response: SendResponse and SendRedirect. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

Generating the Content of Response Messages

Web Broker provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from TCustomContentProducer or one of its descendants.

TCustomContentProducer provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in Using page producer components.
- Table producers create HTML commands based on the information in a dataset. They are described in Using database information in responses.

Using Page Producer Components

Page producers (TPageProducer and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

HTML Templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName=Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

When working with HTML templates, you will

- Optionally, Use predefined HTML-transparent tag Names
- Specify the HTML template
- Convert HTML-transparent tags

Using Predefined HTML-transparent Tag Names

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the TTag data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
Link	*tgLink*	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
Image	*tgImage*	A graphic image. The result is an HTML <IMG> tag.
Table	*tgTable*	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
ImageMap	*tgImageMap*	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.
Object	*tgObject*	An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag.
Embed	*tgEmbed*	A Netscape-compliant add-in DLL. The result is an HTML sequence beginning with an <EMBED> tag and ending with an </EMBED> tag.

Any other tag name is associated with tgCustom. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

Note: The predefined tag names are case insensitive.

Specifying the HTML Template

Page producers provide you with many choices in how to specify the HTML template. You can set the HTMLFile property to the name of a file that contains the HTML template. You can set the HTMLDoc property to a TStrings

object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the Content method.

In addition, you can call the ContentFromString method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the ContentFromStream method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a TBlobStream object.

Converting HTML-transparent Tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the OnHTMLTag event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content. See Using page producers from an action item for a simple example of converting HTML-transparent tags.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

Using Page Producers from an Action Item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
[Delphi]
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
   PageProducer1.HTMLFile := 'Greeting.html';
   Response.Content := PageProducer1.Content;
end;
```

```
[C++]
void __fastcall WebModule1::MyActionEventHandler(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  PageProducer1->HTMLFile = "Greeting.html";
  Response->Content = PageProducer1->Content();
}
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our Web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (`<#UserName>`) in the HTML during execution:

```
[Delphi]
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
```

```
    if CompareText(TagString,'UserName') = 0 then
       ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

```
[C++]
void __fastcall WebModule1::HTMLTagHandler(TObject *Sender, TTag Tag,
   const AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
  if (CompareText(TagString,"UserName") == 0)
    ReplaceText = ((TPageProducer *)Sender)->Dispatcher->Request->Content;
}
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our Web site.
</BODY>
</HTML>
```

Note: This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

Chaining Page Producers Together

The replacement text from an OnHTMLTag event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the ContentFromString method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
[Delphi]
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

```
[C++]
void __fastcall WebModule1::Action2Action(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer2->ContentFromString(Response->Content);
}
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the

month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:

The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=January Year=2000>.
- Replaces <#TitleLine> with <#Calendar Month=December Year=1999 Size=Small> January 2000 <#Calendar Month=February Year=2000 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=2000 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=January Year=2000> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the #Calendar tags with appropriate HTML tables.

Using Database Information in Responses

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

To return database information in an HTTP response, you would typically

- Add a session to the Web module
- Represent the database information in HTML

As an alternate approach, special components on the InternetExpress category of the **Tool palette** let you build Web servers that are part of a multi-tiered database application. See Building Web applications using InternetExpress for details.

Adding a Session to the Web Module

Console CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session.

When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session.

Each request message in an ISAPI or NSAPI application spawns a new thread. The Web module for that thread is generated dynamically at runtime. Add a *TSession* object to the Web module to handle the database connections for the thread that contains the Web module.

Separate instances of the Web module are generated for each thread at runtime. Each of those modules contains the session object. Each of those sessions must have a separate name, so that the threads that handle separate request messages do not interfere with each other's database connections. To cause the session objects in each module to dynamically generate unique names for themselves, set the AutoSessionNameproperty of the session object. Each session object will dynamically generate a unique name for itself and set the SessionName property of all datasets in the module to refer to that unique name. This allows all interaction with the database for each request thread to proceed without interfering with any of the other request messages. For more information on sessions, see Managing database sessions.

Representing a Dataset in HTML

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

Using Dataset Page Producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tag name which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see Using page producer components.

To use a dataset page producer, add a TDataSetPageProducer component to your Web module and set its DataSet property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

Using Table Producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

- Specifying the table attributes
- Specifying the row attributes
- Specifying the columns
- Embedding tables in HTML documents

Specifying the Table Attributes

Table producers use the THTMLTableAttributes object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the **Object Inspector**. Select the table producer object in the **Object Inspector** and expand the TableAttributes property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

Specifying the Row Attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The RowAttributes property is a THTMLTableRowAttributes object.

At design time, specify these properties using the **Object Inspector** by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the MaxRows property.

Specifying the Columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the **Object Inspector** after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate THTMLTableColumn objects and using the methods of the Columns property to add them to the table. If you do not set up the *Columns property*, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

Embedding Tables in HTML Documents

You can embed the HTML table that represents your dataset in a larger document by using the Header and Footer properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the Caption and CaptionAlignment properties to give your table a caption.

Using TDataSetTableProducer

TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the DataSet property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

Using TQueryTableProducer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the TQuery object that uses those parameters as the Query property of a TQueryTableProducer component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the Content method of *TQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

Using WebSnap

Creating Web Server Applications Using WebSnap

WebSnap augments Web Broker with additional components, wizards, and views—making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers.

WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance. The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a simple text editor. Changes can be made to the templates as needed, even after the application is deployed. There is no need to modify the project source code at all, which saves valuable development time. Also, WebSnap's multiple module support can be used to partition your application into smaller pieces during the coding phases of your project. Developers can work more independently.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. WebSnap components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make dataset components scriptable. You can use WebSnap producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked . The page module views show the result of server-side scripting without running the application. You can view the generated HTML in text form using the HTML Result tab. The HTML Script tab shows the page with server-side scripting, which is used to generate HTML for the page.

The following topics explain how to use the WebSnap components to create a Web server application:

- Fundamental WebSnap components
- Creating Web Server Applications
- Server-side scripting in WebSnap
- Dispatching requests

Fundamental WebSnap Components

Before you can build Web server applications using WebSnap, you must first understand the fundamental components used in WebSnap development. They fall into three categories:

- Web modules, which contain the components that make up the application and define pages
- Adapters, which provide an interface between HTML pages and the Web server application itself
- Page producers, which contain the routines that create the HTML pages to be served to the end user

The following sections examine each type of component in more detail.

Web Modules

Web modules are the basic building block of WebSnap applications. Every WebSnap server application must have at least one Web module. More can be added as needed. There are four Web module types:

- Web application page modules (TWebAppPageModule objects)
- *Web application data modules* (TWebAppDataModule objects)
- *Web page modules (*TWebPageModule objects)
- *Web data modules (*TWebDataModule objects)

Web page modules and Web application page modules provide content for Web pages. Web data modules and Web application data modules act as containers for components shared across your application; they serve the same purpose in WebSnap applications that ordinary data modules serve in regular applications. You can include any number of Web page or data modules in your server application.

You may be wondering how many Web modules your application needs. Every WebSnap application needs one (and only one) Web application module of some type. Beyond that, you can add as many Web page or data modules as you need.

For Web page modules, a good rule of thumb is one per page style. If you intend to implement a page that can use the format of an existing page, you may not need a new Web page module. Modifications to an existing page module may suffice. If the page is very different from your existing modules, you will probably want to create a new module. For example, let's say you are trying to build a server to handle online catalog sales. Pages which describe available products might all share the same Web page module, since the pages can all contain the same basic information types using the same layout. An order form, however, would probably require a different Web page module, since the format and function of an order form is different from that of an item description page.

The rules are different for Web data modules. Components that can be shared by many different Web modules should be placed in a Web data module to simplify shared access. You will also want to place components that can be used by many different Web applications in their own Web data module. That way you can easily share those items among applications. Of course, if neither of these circumstances applies you might choose not to use Web data modules at all. Use them the same way you would use regular data modules, and let your own judgment and experience be your guide.

The following topics describe Web modules in greater detail:

- Web application module types
- Web page modules
- Web data modules

Web Application Module Types

Web application modules provide centralized control for business rules and non-visual components in the Web application. The two types of Web application modules are tabulated below.

Web application module types

Web application module type	Description
Page	Creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
Data	Used as a container for components shared by other modules, such as database components used by multiple Web page modules.

Web application modules act as containers for components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. If you are already familiar with the Web Broker architecture, you can think of Web application modules as being similar to *TWebApplication* objects. Web application modules also contain the functionality of a regular Web module, either page or data, depending on the Web application module type. Your project can contain only one Web application module. You will never need more than one anyway; you can add regular Web modules to your server to provide whatever extra features you want.

Use the Web application module to contain the most basic features of your server application. If your server will maintain a home page of some sort, you may want to make your Web application module a *TWebAppPageModule* instead of a *TWebAppDataModule*, so you don't have to create an extra Web page module for that page.

Web Page Modules

Each Web page module has a page producer associated with it. When a request is received, the page dispatcher analyzes the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. A Web page module is more than a mere container, however. A Web page module is used specifically to produce a Web page.

Page producer component

Web page modules have a property that identifies the page producer component responsible for generating content for the page. The WebSnap page module wizard automatically adds a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap category . However, if the page module has a template file, be sure that the content of this file is compatible with the replacement producer component.

Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module's unit specifies the page name for the Web page module.

Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side scripting. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser.

(The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.)

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the **Project Manager** and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file, the properties of the page producer component specify the template.

Web Data Modules

Like standard data modules, Web data modules are a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

You can also use Web data modules to contain sets of components that can be used by several different Web server applications.

Structure of a Web data module unit

Standard data modules have a variable called a form variable, which is used to access the data module object. Web data modules replace the variable with a function, which is defined in a Web data module's unit and has the same name as the Web data module. The function's purpose is the same as that of the variable it replaces. WebSnap applications may be multi-threaded and may have multiple instances of a particular module to service multiple requests concurrently. Therefore, the function is used to return the correct instance.

The Web data module unit also registers a factory to specify how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests or to destroy the module after a request has been serviced.

Adapters

Adapters define a script interface to your server application. They allow you to insert scripting languages into a page and retrieve information by making calls from your script code to the adapters. For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields. This is similar to the transparent tags used in Web Broker applications. Adapters also support actions that execute commands. For example, clicking on a hyperlink or submitting an HTML form can initiate adapter actions.

Adapters simplify the task of creating HTML pages dynamically. By using adapters in your application, you can include object-oriented script that supports conditional logic and looping. Without adapters and server-side scripting, you must write more of your HTML generation logic in event handlers. Using adapters can significantly reduce development time.

See Server-side scripting in WebSnap for more details about scripting.

Four types of adapter components can be used to create page content: fields, actions, errors and records.

Fields

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image

written to the Web page. When a page displays its content, a request is sent to the Web server application, which invokes the adapter dispatcher to retrieve the actual image from the field component.

Actions

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

Errors

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

Records

Some adapter components, such as *TDataSetAdapter*, represent multiple records. The adapter provides a scripting interface which allows iteration through the records. Some adapters support paging and iterate only through the records on the current page.

Page Producers

Page producers to generate content on behalf of a Web page module. Page producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLFile property, or the internal string using the HTMLDoc property.
- When the producers are used with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML that can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. To learn more about using transparent tags, see Converting HTML-transparent tags. Active scripting support allows you to embed JScript or VBScript inside the HTML page.

The standard WebSnap method for using page producers is as follows. When you create a Web page module, you must choose a page producer type in the Web Page Module wizard. You have many choices, but most WebSnap developers prototype their pages by using an adapter page producer, TAdapterPageProducer. The adapter page producer lets you build a prototype Web page using a process analogous to the standard component model. You add a type of form, an adapter form, to the adapter page producer. As you need them, you can add adapter components (such as adapter grids) to the adapter form. Using adapter page producers, you can create Web pages in a way that is similar to the standard technique for building user interfaces.

There are some circumstances where switching from an adapter page producer to a regular page producer is more appropriate. For example, part of the function of an adapter page producer is to dynamically generate script in a page template at runtime. You may decide that static script would help optimize your server. Also, users who are experienced with script may want to make changes to the script directly. In this case, a regular page producer must be used to avoid conflicts between dynamic and static script. To learn how to change to a regular page producer, see the Advanced HTML design topic.

You can also use page producers the same way you would use them in Web Broker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are

- the ability to preview the page's layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

Creating Web Server Applications with WebSnap

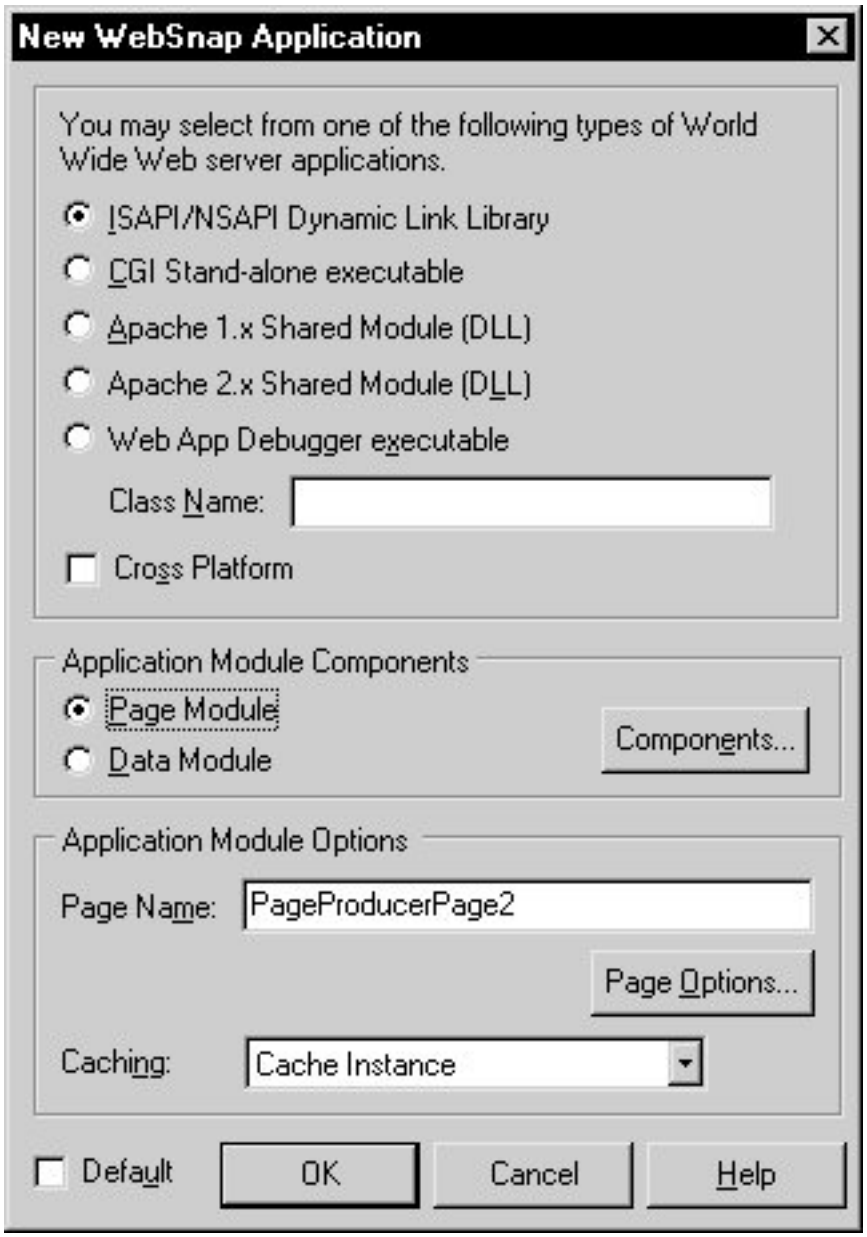
If you look at the source code for WebSnap, you will discover that WebSnap comprises hundreds of objects. In fact, WebSnap is so rich in objects and features that you could spend a long time studying its architecture in detail before understanding it completely. Fortunately, you really don't need to understand the whole WebSnap system before you start developing your server application.

Here you will learn more about how WebSnap works by creating a new Web server application.

To create a new Web server application using the WebSnap architecture:

1. Choose **File** ▸ **New** ▸ **Other**, and select the WebSnap folder from Delphi Projects.
2. In the right pane of the New Items window choose WebSnap Application.

   A dialog box appears (as shown below)
3. Specify the correct server type.
4. Use the components button to specify application module components.
5. Use the Page Options button to select application module options.



For further information about adding application module components, see Specifying Application Module Components.

Selecting a Server Type

Select one of the following types of Web server application, depending on your application's type of Web server.

Web server application types

Server type	Description
ISAPI and NSAPI	Sets up your project as a DLL with the exported methods expected by the Web server.
Apache	Sets up your project as a DLL with the exported methods expected by the appropriate Apache Web server. Both Apache 1 and 2 are supported.
CGI stand-alone	Sets up your project as a console application which conforms to the Common Gateway Interface (CGI) standard.

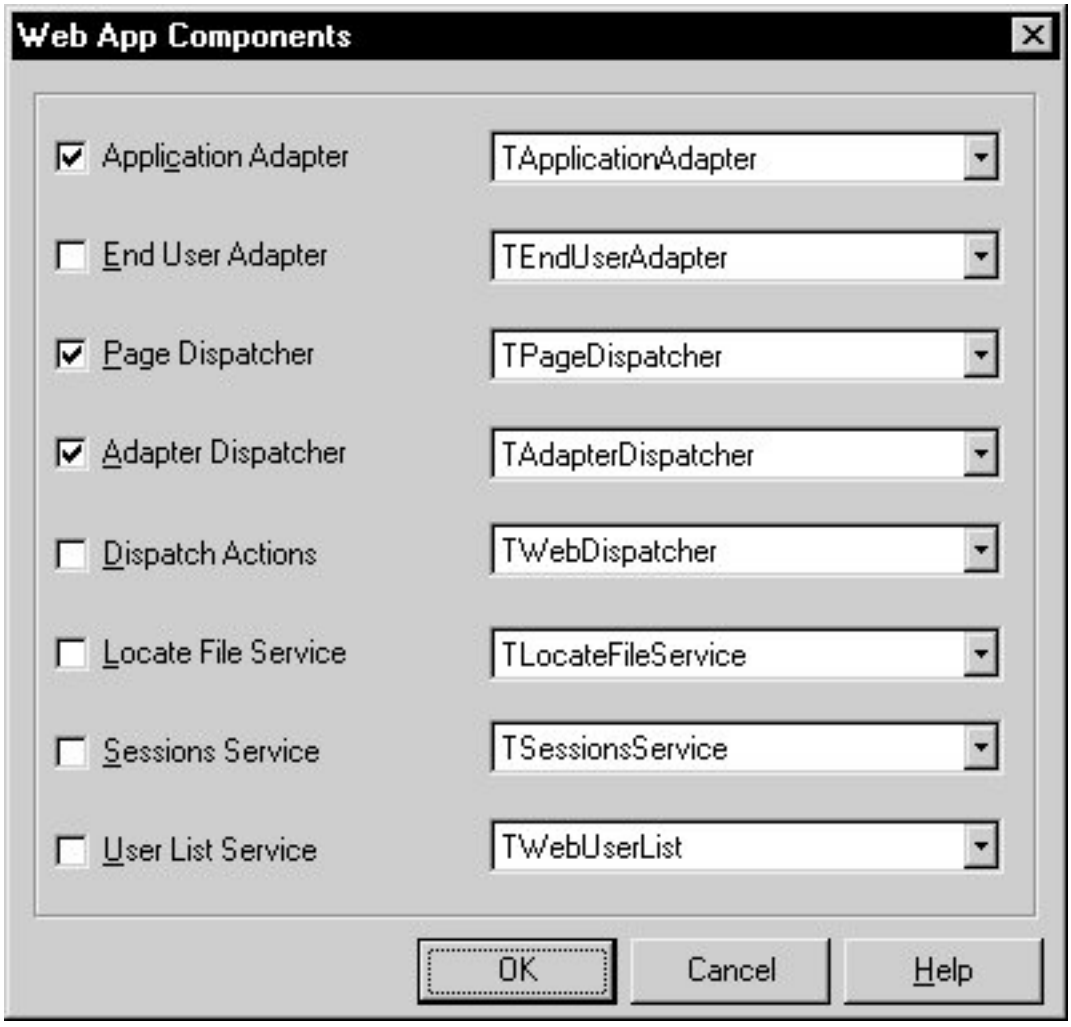
Web App Debugger executable	Creates an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Specifying Application Module Components

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

For information on creating web server applications, see Creating Web Server Applications with WebSnap.

Selecting the Components button on the new WebSnap application dialog displays another dialog that allows you to select one or more of the Web application module components. The dialog, which is called the Web App Components dialog, is shown below.



The following table contains a brief explanation of the available components:

Web application components

Component type	Description
Application Adapter	Contains information about the application, such as the title. The default type is *TApplicationAdapter*.

End User Adapter	Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
Page Dispatcher	Examines the HTTP request's pathinfo and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
Adapter Dispatcher	Automatically handles HTML form submissions and requests for dynamic images by calling adapter action and field components. The default type is *TAdapterDispatcher*.
Dispatch Actions	Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events or request the content of page-producer components. The default type is *TWebDispatcher*.
Locate File Service	Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
Sessions Service	Stores information about end users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users and to automatically log a user out after a period of inactivity. The default type is *TSessionsService*.
User List Service	Keeps track of authorized users, their passwords, and their access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types shipped with the IDE. Users can create their own component types or use third-party component types.

For information about modifying application module components, see Selecting Web Application Module Options.

Selecting Web Application Module Options

If the selected application module type is a page module, you can associate a name with the page by entering a name in the Page Name field in the New WebSnap Application dialog box. At runtime, the instance of this module can be either kept in cache or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options by choosing the Page Options button.

For information on adding application module components, see Specifying Application Module Components.

The Application Module Page Options dialog is displayed and provides the following categories:

Note: The *AdapterPageProducer* supports only JScript.

- Producer: The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.
- HTML: When the selected producer uses an HTML template this group will be visible.
- XSL: When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- New File: Check New File if you want a template file to be created and managed as part of the unit. A managed template file appears in the **Project Manager** and has the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- Template: When New File is checked, choose the default content for the template file from the Template drop-down. The standard template displays the title of the application, the title of the page, and hyperlinks to published pages. The blank template creates a blank page.
- Page: Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic, whereas the title is the name that the end user will see when the page is displayed in a browser.
- Published: Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.

- Login Required: Check Login Required to require the user to log on before the page can be accessed.

Advanced HTML Design

Using adapters and adapter page producers, WebSnap makes it easy to create scripted HTML pages in your Web server application. You can create a Web front end for your application data using WebSnap tools that may suit all of your needs. One powerful feature of WebSnap, however, is the ability to incorporate Web design expertise from other sources into your application. This section discusses some strategies for expanding the Web server design and maintenance process to include other tools and non-programmer team members.

The end products of WebSnap development are your server application and HTML templates for the pages that the server produces. The templates include a mixture of scripting and HTML. Once they have been generated initially, they can be edited at any time using any HTML tool you like. (It would be best to use a tool that supports embedded script tags, like Microsoft FrontPage, to ensure that the editor doesn't accidentally damage the script.) The ability to edit template pages outside of the IDE can be used many ways.

After the product has been deployed, you may wish to change the look of the final HTML pages. Perhaps your software development team is not even responsible for the final page layout. That duty may belong to a dedicated Web page designer in your organization, for example. Your page designers may not have any experience with software development. Fortunately, they don't have to. They can edit the page templates at any point in the product development and maintenance cycle, without ever changing the source code. Thus, WebSnap HTML templates can make server development and maintenance more efficient.

Manipulating server-side script in HTML files

HTML in page templates can be modified at any time in the development cycle. Server-side scripting can be a different matter, however. It is always possible to manipulate the server-side script in the templates outside of the IDE, but it is not recommended for pages generated by an adapter page producer. The adapter page producer is different from ordinary page producers in that it can change the server-side scripting in the page templates at runtime. It can be difficult to predict how your script will act if other script is added dynamically. If you want to manipulate script directly, make sure that your Web page module contains a page producer instead of an adapter page producer.

If you have a Web page module that uses an adapter page producer, you can convert it to use a regular page producer instead by using the following steps.

To modify a Web page module to use a regular page producer

1. You can access the page module view with server-side scripting using the HTML Script tab. In the module you want to convert (let's call it ModuleName), copy all of the information from the HTML Script tab to the ModuleName.html tab, replacing all of the information that it contained previously.

   Note: When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked .

2. Drop a page producer (located on the Internet category of the **Tool Palette**) onto your Web page module.
3. Set the page producer's ScriptEngine property to match that of the adapter page producer it replaces.
4. Change the page producer in the Web page module from the adapter page producer to the new page producer.
5. The adapter page producer has now been bypassed. You may now delete it from the Web page module.

Login Support

Many Web server applications require login support. For example, a server application may require a user to login before granting access to some parts of a Web site. Pages may have a different appearance for different users; logins may be necessary to enable the Web server to send the right pages. Also, because servers have physical limitations on memory and processor cycles, server applications sometimes need the ability to limit the number of users at any given time.

With WebSnap, incorporating login support into your Web server application is fairly simple and straightforward. You can add login support, either by designing it in from the beginning of your development process or by retrofitting it onto an existing application.

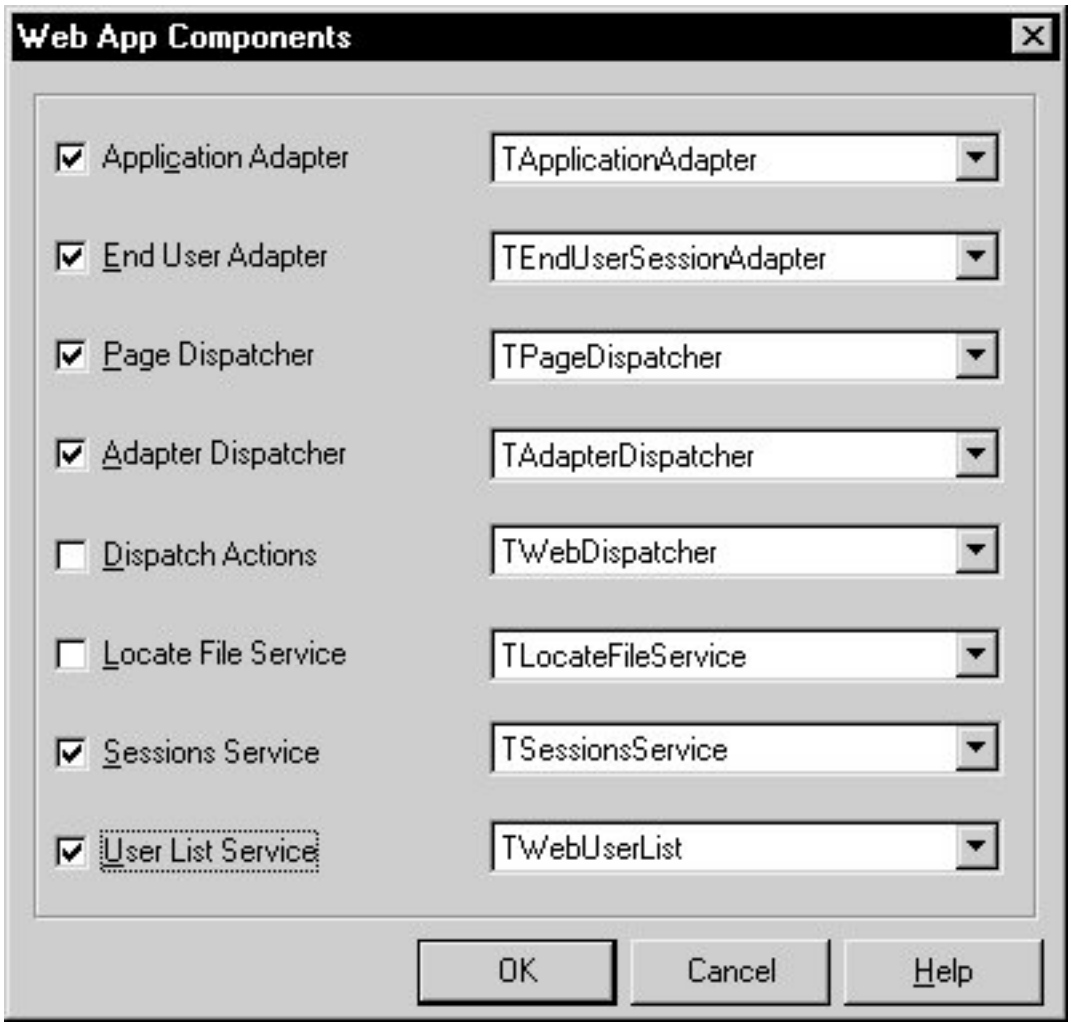
For additional information on adding login support, refer to Adding Login Support.

Adding Login Support

In order to implement login support, you need to make sure your Web application module has the following components:

- A user list service (an object of type TWebUserList), which contains the usernames, passwords and permissions for server users
- A sessions service (TSessionsService), which stores information about users currently logged in to the server
- An end user adapter (TEndUserSessionAdapter) which handles actions associated with logging in

When you first create your Web server application, you can add these components using the New WebSnap Application dialog box. Click the Components button on that dialog to display the New Web App Components dialog box. Check the End User Adapter, Sessions Service and Web User List boxes. Select *TEndUserSessionAdapter* on the drop down menu next to the End User Adapter box to select the end user adapter type. (The default choice, TEndUserSessionAdapter, is not appropriate for login support because it cannot track the current user.) When you're finished, your dialog should look like the one shown below. Click OK twice to dismiss the dialog boxes. Your Web application module now has the necessary components for login support.



If you are adding login support to an existing Web application module, you can drag these components directly into your module from the WebSnap category of the **Tool Palette**. The Web application module will configure itself automatically.

The sessions service and the end user adapter may not require your attention during your design phase, but the Web user list probably will. You can add default users and set their read/modify permissions through the WebUserList component editor. Double-click on the component to display an editor which lets you set usernames, passwords and access rights. For more information on how to set up access rights, see the topic "User access rights".

For information on login support, see Login Support.

Using the Sessions Service

The sessions service, which is an object of type TSessionsService, keeps track of the users who are logged into your Web server application. The sessions service is responsible for assigning a different session for each user and for associating name/value pairs (such as a username) with a user.

Information contained in a sessions service is stored in the application's memory. Therefore, the Web server application must keep running between requests for the sessions service to work. Some server application types, such as CGI, terminate between requests.

Note: If you want your application to support logins, be sure to use a server type that does not terminate between requests. If your project produces a Web App debugger executable, you must have the application running

in the background before it receives a page request. Otherwise it will terminate after each page request, and users will never be able to get past the login page.

There are two important properties in the sessions service which you can use to change default server behavior. The *MaxSessions* property specifies how many users can be logged into the system at any given time. The default value for *MaxSessions* is -1, which places no software limitation on the number of allowed users. Of course, your server hardware can still run short of memory or processor cycles for new users, which can adversely affect system performance. If you are concerned that excessive numbers of users might overwhelm your server, be sure to set *MaxSessions* to an appropriate value.

The *DefaultTimeout* property specifies the defaut time-out period in minutes. After *DefaultTimeout* minutes have passed without any user activity, the session is automatically terminated. If the user had logged in, all login information is lost.. The default value is 20. You can override the default value in any given session by changing its *TimeoutMinutes* property.

Login Pages

Of course, your Websnap application also needs a login page. Users enter their username and password for authentication, either while trying to access a restricted page or prior to such an attempt. The user can also specify which page they receive when authentication is completed. If the username and password match a user in the Web user list, the user acquires the appropriate access rights and is forwarded to the page specified on the login page. If the user isn't authenticated, the login page may be redisplayed (the default action) or some other action may occur.

Fortunately, WebSnap makes it easy to create a simple login page using a Web page module and the adapter page producer. To create a login page, start by creating a new Web page module. Choose **File** ▸ **New** ▸ **Other,** and select WebSnap from the Delphi Projects folder. In the right pane of the New Items window select the WebSnap Page Module. Select AdapterPageProducer as the page producer type. Fill in the other options however you like. Login tends to be a good name for the login page.

Now you should add the most basic login page fields: a username field, a password field, a selection box for selecting which page the user receives after logging in, and a Login button which submits the page and authenticates the user.

To add these fields:

1. Add a TLoginFormAdapter component (which you can find on the WebSnap category of the **Tool Palette**) to the Web page module you just created.
2. Double-click the *AdapterPageProducer* component to display a Web page editor window.
3. Right-click the *AdapterPageProducer* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterForm* and click OK.
4. Add an *AdapterFieldGroup* to the AdapterForm. (Right-click the *AdapterForm* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterFieldGroup* and click OK.)
5. Now go to the **Object Inspector** and set the *Adapter* property of your *AdapterFieldGroup* to your *LoginFormAdapter*. The UserName, Password and NextPage fields should appear automatically in the Browser tab of the Web page editor (accessed by double clicking the *AdapterPageProducer*) .

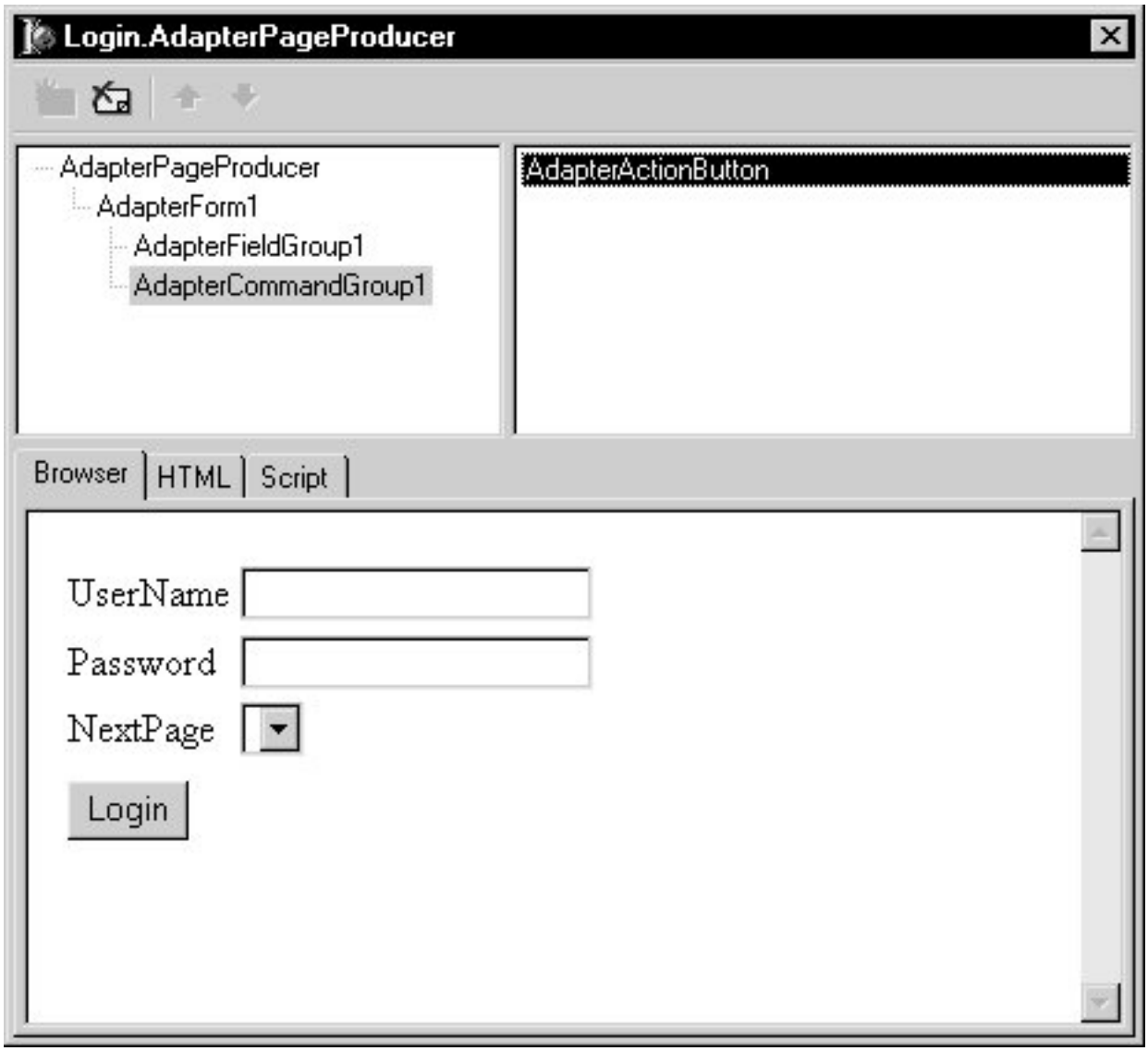
So, WebSnap takes care of most of the work in a few simple steps. The login page is still missing a Login button, which submits the information on the form for authentication.

To add a Login button:

1. Add an *AdapterCommandGroup* to the *AdapterForm*.
2. Add an *AdapterActionButton* to the *AdapterCommandGroup*. Change its *DisplayComponent* to AdapterFieldGroup using the **Object Inspector**.

3 Click on the *AdapterActionButton* (listed in the upper right pane of the Web page editor) and change its *ActionName* property to Login using the **Object Inspector**. You can see a preview of your login page in the Web page editor's Browser tab.

Your Web page editor should look similar to the one shown below.



If the button doesn't appear below the *AdapterFieldGroup*, make sure that the *AdapterCommandGroup* is listed below the *AdapterFieldGroup* on the Web page editor. If it appears above, select the *AdapterCommandGroup* and click the down arrow on the Web page editor. (In general, Web page elements appear vertically in the same order as they appear in the Web page editor.)

There is one more step necessary before your login page becomes functional. You need to specify which of your pages is the login page in your end user session adapter. To do so, select the *EndUserSessionAdapter* component in your Web application module. In the **Object Inspector**, change the *LoginPage* property to the name of your login page. Your login page is now enabled for all the pages in your Web server application.

Setting Pages to Require Logins

Once you have a working login page, you must require logins for those pages which need controlled access. The easiest way to have a page require logins is to design that requirement into the page. When you first create a Web page module, check the Login Required box in the Page section of the New WebSnap Page Module dialog box.

If you create a page without requiring logins, you can change your mind later.

To require logins after a Web page module has been created:

1. Open the source code file associated with the Web page module in the editor.
2. Scroll down to the implementation section. In the parameters for the WebRequestHandler.AddWebModuleFactory command, find the creator of the TWebPageInfo object. It should look like this:

```
[Delphi]
TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html')
```

3. Uncomment the *wpLoginRequired* portion of the parameter list by removing the curly braces. The TWebPageInfo creator should now look like this:

```
[Delphi]
TWebPageInfo.Create([wpPublished , wpLoginRequired], '.html')
```

```
[C++]
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished << wpLoginRequired, ".html", "", "", "", "");
```

To remove the login requirement from a page, reverse the process and recomment the *wpLoginRequired* portion of the creator.

Note: You can use the same process to make the page published or not. Simply add or remove comment marks around the *wpPublished* portion as needed.

User Access Rights

User access rights are an important part of any Web server application. You need to be able to control who can view and modify the information your server provides. For example, let's say you are building a server application to handle online retail sales. It makes sense to allow users to view items in your catalog, but you don't want them to be able to change your prices! Clearly, access rights are an important issue.

Fortunately, WebSnap offers you several ways to control access to pages and server content. In previous topics, you saw how you can control page access by requiring logins. You have other options as well. For example:

- You can show data fields in an edit box to users with appropriate modify access rights; other users will see the field contents, but not have the ability to edit them.
- You can hide specific fields from users who don't have the correct view access rights.
- You can prevent unauthorized users from receiving specific pages.

Descriptions for implementing these behaviors are included in this topic.

Dynamically displaying fields as edit or text boxes

If you use the adapter page producer, you can change the appearance of page elements for users with different access rights. For example, the Biolife demo (found in the WebSnap subdirectory of the Demos directory) contains a form page which shows all the information for a given species. The form appears when the user clicks a Details button on the grid. A user logged in as Will sees data displayed as plain text. Will is not allowed to modify the data, so the form doesn't give him a mechanism to do so. User Ellen does have modify permissions, so when Ellen views the form page, she sees a series of edit boxes which allow her to change field contents. Using access rights in this manner can save you from creating extra pages.

The appearance of some page elements, such as *TAdapterDisplayField*, is determined by its *ViewMode* property. If *ViewMode* is set to *vmToggleOnAccess*, the page element will appear as an edit box to users with modify access. Users without modify access will see plain text. Set the *ViewMode* property to *vmToggleOnAccess* to allow the page element's appearance and function to be determined dynamically.

A Web user list is a list of *TWebUserListItem* objects, one for each user who can login to the system. Permissions for users are stored in their Web user list item's *AccessRights* property. *AccessRights* is a text string, so you are free to specify permissions any way you like. Create a name for every kind of access right you want in your server application. If you want a user to have multiple access rights, separate items in the list with a space, semicolon or comma.

Access rights for fields are controlled by their *ViewAccess* and *ModifyAccess* properties. *ViewAccess* stores the name of the access rights needed to view a given field. *ModifyAccess* dictates what access rights are needed to modify field data. These properties appear in two places: in each field and in the adapter object that contains them.

Checking access rights is a two-step process. When deciding the appearance of a field in a page, the application first checks the field's own access rights. If the value is an empty string, the application then checks the access rights for the adapter which contains the field. If the adapter property is empty as well, the application will follow its default behavior. For modify access, the default behavior is to allow modifications by any user in the Web user list who has a non-empty *AccessRights* property. For view access, permission is automatically granted when no view access rights are specified.

Hiding fields and their contents

You can hide the contents of a field from users who don't have appropriate view permissions. First set the *ViewAccess* property for the field to match the permission you want users to have. Next, make sure that the *ViewAccess* for the field's page element is set to *vmToggleOnAccess*. The field caption will appear, but the value of the field won't.

Of course, it is often best to hide all references to the field when a user doesn't have view permissions. To do so, set the *HideOptions* for the field's page element to include *hoHideOnNoDisplayAccess*. Neither the caption nor the contents of the field will be displayed.

Preventing page access

You may decide that certain pages should not be accessible to unauthorized users. To grant check access rights before displaying pages, alter your call to the *TWebPageInfo* constructor in the Web request handler's AddWebModuleFactory command. This command appears in the initialization section of the source code for your module.

The constructor for TWebPageInfo takes up to 6 arguments. WebSnap usually leaves four of them set to default values (empty strings), so the call generally looks like this:

```
[Delphi]
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html')
```

```
[C++]
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "");
```

To check permissions before granting access, you need to supply the string for the necessary permission in the sixth parameter. For example, let's say that the permission is called "Access". This is how you could modify the creator:

```
[Delphi]
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html', '', '', '', 'Access')
```

```
[C++]
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "Access");
```

Access to the page will now be denied to anyone who lacks Access permission.

Server-side Scripting in WebSnap

Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web server application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

This topic provides a conceptual overview of server-side scripting and how it is used by WebSnap applications. Although server-side scripting is a valuable part of WebSnap, it is not essential that you use scripting in your WebSnap applications. Scripting is used for HTML generation and nothing else. It allows you to insert application data into an HTML page. In fact, almost all of the properties exposed by adapters and other script-enabled objects are read-only. Server-side script isn't used to change application data, which is still managed by components and event handlers written in your application's source code.

There are other ways to insert application data into an HTML page. You can use Web Broker's transparent tags or some other tag-based solution, if you prefer. For example, several projects in the WebSnap examples directory use XML and XSL instead of scripting. Without scripting, however, you will be forced to write most of your HTML generation logic in source code, which will increase your development time.

The scripting used in WebSnap is object-oriented and supports conditional logic and looping, which can greatly simplify your page generation tasks. For example, your pages may include a data field that can be edited by some users but not others. With scripting, conditional logic can be placed in your template pages which displays an edit box for authorized users and simple text for others. With a tag-based approach, you must program such decision-making into your HTML generating source code.

Active scripting

WebSnap relies on *active scripting* to implement server-side script. Active scripting is a technology created by Microsoft to allow a scripting language to be used with application objects through COM interfaces. Microsoft ships two active scripting languages, VBScript and JScript. Support for other languages is available through third parties.

Script engine

The page producer's *ScriptEngine* property identifies the active scripting engine that evaluates the script within a template. It is set to support JScript by default, but it can also support other scripting languages (such as VBScript).

Note: WebSnap's adapters are designed to produce JScript. You will need to provide your own script generation logic for other scripting languages.

Script blocks

Script blocks, which appear in HTML templates, are delimited by <% and %>. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JScript block generates a list of five numbered lines:

```
<ul>
<% for (i=0;i<5;i++) { %>
   <li>Item <%=i %></li>
<% } %>
</ul>
```

(The <%= delimiter is short for *Response.Write*.)

Creating script

Developers can take advantage of WebSnap features to automatically generate script.

Wizard templates

When creating a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the Default template generates JScript which, in turn, displays the application title, page name, and links to published pages.

TAdapterPageProducer

The TAdapterPageProducer builds forms and tables by generating HTML and JScript. The generated JScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

Editing and viewing script

When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked. You can access the page module view with the HTML resulting from the executed script using the HTML Script tab. The HTML Script tab displays the HTML and JScript generated by the *TAdapterPageProducer* object. Consult this view to see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

Including script in a page

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page. Use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

Script Objects

Script objects are objects that script commands can reference. You make objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

Script objects

Script object	Description
Application	Provides access to the application adapter of the Web Application module.
EndUser	Provides access to the end user adapter of the Web Application module.
Session	Provides access to the session object of the Web Application module.
Pages	Provides access to the application pages.
Modules	Provides access to the application modules.
Page	Provides access to the current page
Producer	Provides access to the page producer of the Web Page module.
Response	Provides access to the WebResponse. Use this object when tag replacement is not desired.
Request	Provides access to the WebRequest.
Adapter objects	All of the adapter components on the current page can be referenced without qualification. Adapters in other modules must be qualified using the Modules objects.

Script objects on the current page, which all use the same adapter, can be referenced without qualification. Script objects on other pages are part of another page module and have a different adapter object. They can be accessed by starting the script object reference with the name of the adapter object. For example,

```
<%= FirstName %>
```

displays the contents of the *FirstName* property of the current page's adapter. The following script line displays the *FirstName* property of Adapter1, which is in another page module:

```
<%= Adapter1.FirstName %>
```

Dispatching Requests and Responses

One reason to use WebSnap for your Web server application development is that WebSnap components automatically handle HTML requests and responses. Instead of writing event handlers for common page transfer chores, you can focus your efforts on your business logic and server design. Still, it can be helpful to understand how WebSnap applications handle HTML requests and responses. This section gives you an overview of that process.

Before handling any requests, the Web application module initializes the Web context object (of type TWebContext). The Web context object, which is accessed by calling the global WebContext function, provides global access to variables used by components servicing the request. For example, the Web context contains the TWebRequest and TWebResponse objects to represent the HTTP request message and the response that should be returned.

The following topics describe Web request handling:

- Using dispatcher components
- Adapter dispatcher operation
- Dispatching action items

- Page dispatcher operation

Dispatcher Components

The dispatcher components in the Web application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher*) looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it passes control to that component.

The Web dispatcher component (*TWebDispatcher*) maintains a collection of action items (of type *TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the *PathInfo* property of the *TWebRequest* object, looking for the name of a registered Web page module. If the dispatcher finds a Web page module name, it passes control to that module.

Adapter Dispatcher Operation

The adapter dispatcher component (*TAdapterDispatcher*) automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Using adapter components to generate content

For WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate the names and values of HTML elements. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form designed to update adapter fields. Typically, page producers use server-side scripting to retrieve names and values from adapter components and then uses this information to generate HTML. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with "Species No=90090" as a parameter. The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

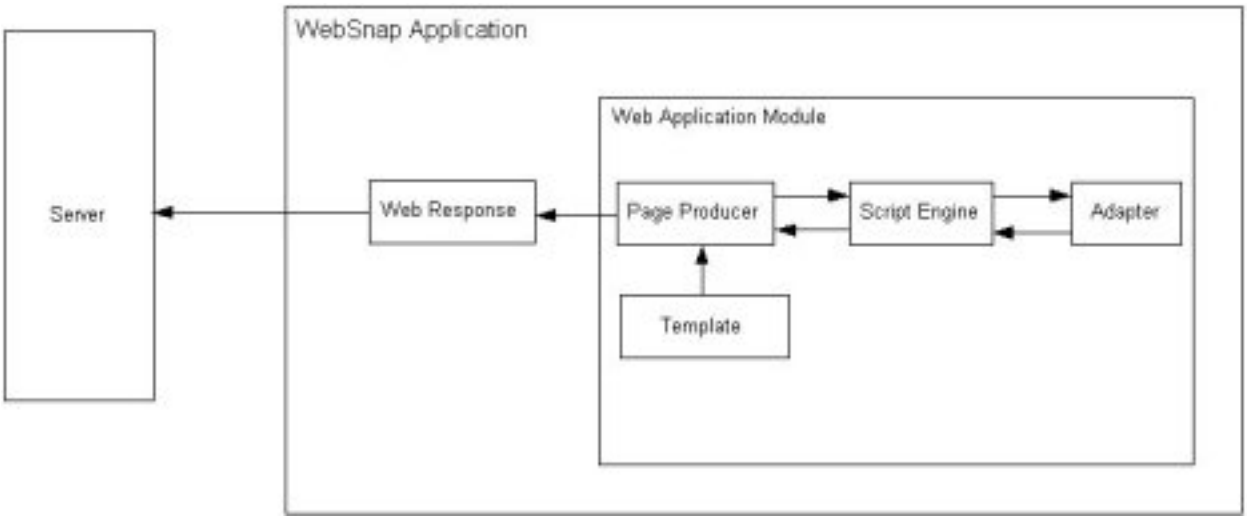
The following script constructs an <A> element referencing the EditRow action of Adapter1 and creates a hyperlink to a page called Details:

```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

The end user clicks this hyperlink, and the browser sends an HTTP request. The adapter dispatcher can determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter Species No=903010. The adapter dispatcher also displays the Edit page if the action executes successfully, and displays the Grid page if action execution fails. It then calls the EditRow action to locate the row to be edited, and the page named Edit is called to generate an HTTP response. The following figure shows how adapter components are used to generate content.



Receiving Adapter Requests and Generating Responses

When the adapter dispatcher receives a client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about that HTTP request. The adapter request and adapter response objects are stored in the Web context to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

Action requests

Action request objects are responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include the following request information:

Request information found in action requests

Request informaton	Description
Component name	Identifies the adapter action component.
Adapter mode	Defines a mode. For example, TDataSetAdapter supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode.
Success page	Identifies the page displayed after successful execution of the action.

Failure page	Identifies the page displayed if an error occurs during execution of the action.
Action request parameters	Identifies the parameters need by the adapter action. For example, the TDataSetAdapter Apply action will include the key values identifying the record to be updated.
Adapter field values	Specifies values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end user, the original values of the adapter field, and uploaded files.
Record keys	Specifies keys that uniquely identify each record.

Generating action responses

Action response objects generate an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting properties within the object, or by calling methods in the action response object. The properties include:

- *RedirectOptions*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *ExecutionStatus*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

The action response methods include:

- *RespondWithPage*—The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web page module directly.

The following figure illustrates how action request and action response objects handle a request.

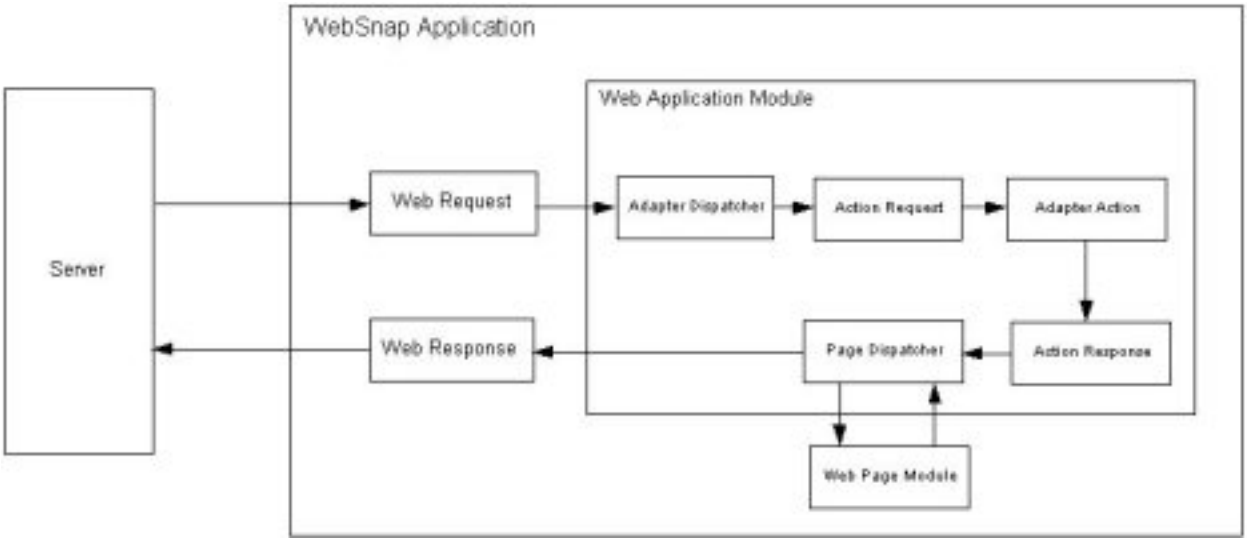


Image request

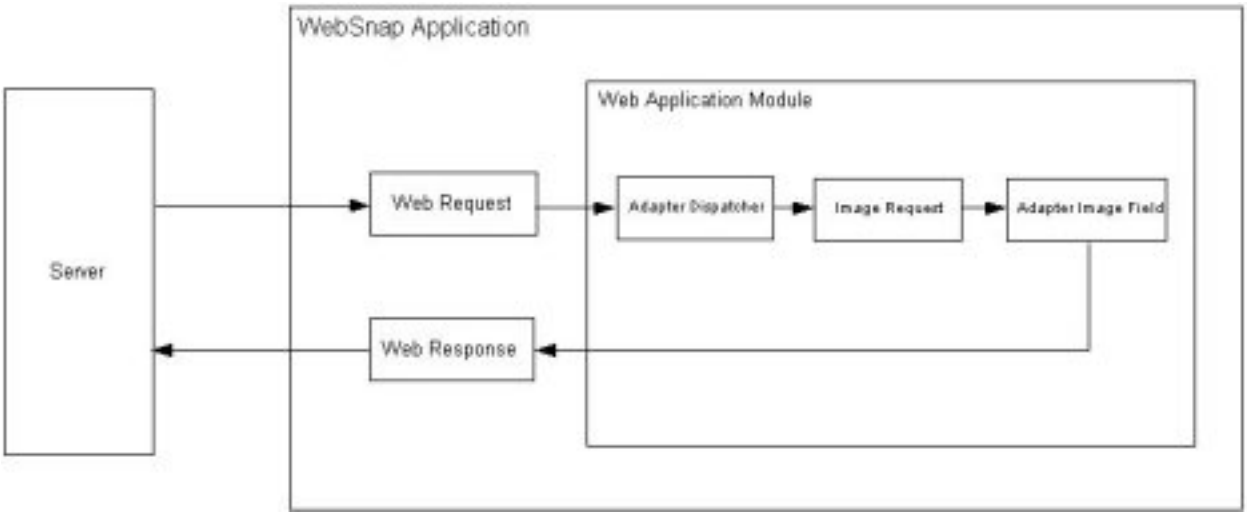
The image request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

- Component name - Identifies the adapter field component.
- Image request parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

Image response

The image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

The following figure illustrates how adapter image fields respond to a request.



Dispatching Action Items

When responding to a request, the Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the PathInfo portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message was not handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. (These components are specific to multi-tiered database applications.) If the request message is still not fully handled, the dispatcher calls

the default action item. The default action item does not need to match either the target URL or the method of the request.

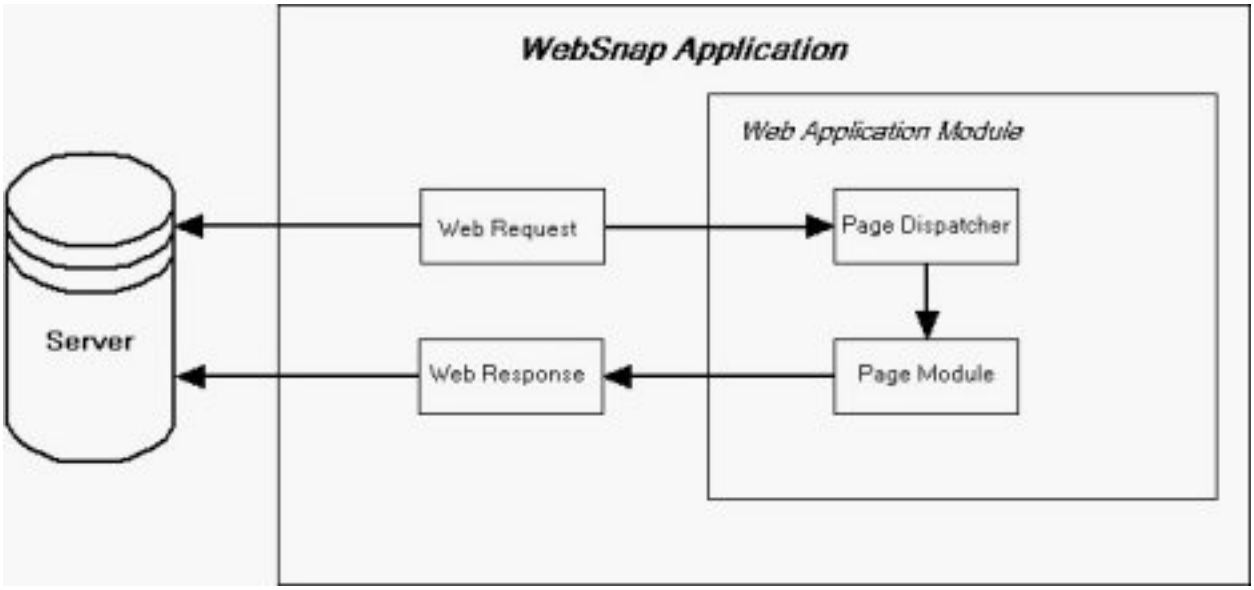
Page dispatcher operation

When the page dispatcher receives a client request, it determines the page name by checking the PathInfo portion of the target URL's request message. If the PathInfo portion is not blank, the page dispatcher uses the ending word of PathInfo as the page name. If the PathInfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank and the Web application module is a page module, the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, it calls that module to generate a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

The following figure shows how the page dispatcher responds to a request.



Using IntraWeb

Creating Web Server Applications Using IntraWeb

IntraWeb is a tool which simplifies Web server application development. You can use IntraWeb to build Web server applications exactly the same way you would build traditional GUI applications, using forms. You can write all of your business logic in the Delphi language; IntraWeb will automatically convert program elements to script or HTML when necessary.

You can use IntraWeb in any of the following modes:

1. **Standalone mode**. IntraWeb uses its own application object type to handle program execution. The application isn't deployed on a commercial server; instead, IntraWeb's own Application Server is used for application deployment.
2. **Application Mode**. The application object is provided by IntraWeb. The application is deployed on a commercial server.
3. **Page mode**. The application object is provided by Web Broker or WebSnap. IntraWeb is used to develop pages. The application is deployed on a commercial server.

IntraWeb applications can be targeted to any of the following server types:

- ISAPI/NSAPI
- Apache versions 1 and 2
- CGI (page mode only)
- Windows services

IntraWeb offers a wide range of browser compatibility. IntraWeb applications automatically detect the user's browser type and generate HTML and script most appropriate for that browser. IntraWeb supports Internet Explorer versions 4 through 6, Netscape 4 and 6, and Mozilla.

Using IntraWeb Components

One of the advantages of IntraWeb is that it uses the same kinds of tools and techniques as regular VCL development. You can build your user interface by dropping components on forms, like you would any other application. There are a number of important differences that you must keep in mind, however. The forms and components used in IntraWeb user interfaces are not the same ones used in non-Web GUI applications. When you create a form or use a component, be sure to use an IntraWeb version instead of a VCL version.

Many VCL components have IntraWeb counterparts. Generally, the IntraWeb components have the same name as their VCL counterparts, with the letters "IW" prefixed to the name. For example, IWCheckBox is the IntraWeb

equivalent of CheckBox. (The name used in source code starts with the letter T, of course, like TIWCheckBox.) On the **Tool palette**, the icons for IntraWeb components are nearly identical to their VCL counterparts, making it easier to find the IntraWeb components you need.

The following table lists VCL components and their IntraWeb counterparts. For more information on these components and how to use them, refer to the IntraWeb help files and other IntraWeb documentation.

VCL and IntraWeb components

VCL component	IntraWeb equivalent	Tool palette category for IntraWeb component
Button	IWButton	IW Standard
CheckBox	IWCheckBox	IW Standard
ComboBox	IWComboBox	IW Standard
DBCheckBox	IWDBCheckBox	IW Data
DBComboBox	IWDBComboBox	IW Data
DBEdit	IWDBEdit	IW Data
DBGrid	IWDBGrid	IW Data
DBImage	IWDBImage	IW Data
DBLabel	IWDBLabel	IW Data
DBListBox	IWDBListBox	IW Data
DBLookupComboBox	IWDBLookupComboBox	IW Data
DBLookupListBox	IWDBLookupListBox	IW Data
DBMemo	IWDBMemo	IW Data
DBNavigator	IWDBNavigator	IW Data
DBText	IWDBText	IW Data
Edit	IWEdit	IW Standard
Image	IWImage or IWImageFile	IW Standard
Label	IWLabel	IW Standard
ListBox	IWListBox	IW Standard
Memo	IWMemo	IW Standard
RadioGroup	IWRadioGroup	IW Standard
Timer	IWTimer	IW Standard
TreeView	IWTreeView	IW Standard

Getting Started with IntraWeb

If you have experience writing GUI applications using Borland's rapid application development tools, then you already have the basic skills you need to start building applications with IntraWeb. The basic design method for the user interface is the same for IntraWeb and regular GUI applications: find the components you need on the **Tool palette** and drop them on a form. Unlike WebSnap's page modules, the appearance of the form mirrors the appearance of the page. The IntraWeb forms and components are distinct from their VCL counterparts, but they are named and arranged similarly.

For example, let's say you want to add a button to a form. In an ordinary VCL application, you would find the Button component on the Standard **Tool palette** category and drop it on your form in an appropriate location. In the compiled application, the button appears where you placed it. For an IntraWeb application, the only difference is that you use

the IWButton component on the IW Standard category. Even the icons for the two different button components look almost identical. The only difference is an "IW" in the top right corner of the IntraWeb button icon.

Follow the four step tutorial, below, to see how easy it is to build an IntraWeb application. The application you develop in the tutorial asks the user for some input and displays the input in a popup window. The tutorial uses IntraWeb's standalone mode, so the application you create will run without a commercial Web server.

The tutorial includes the following steps:

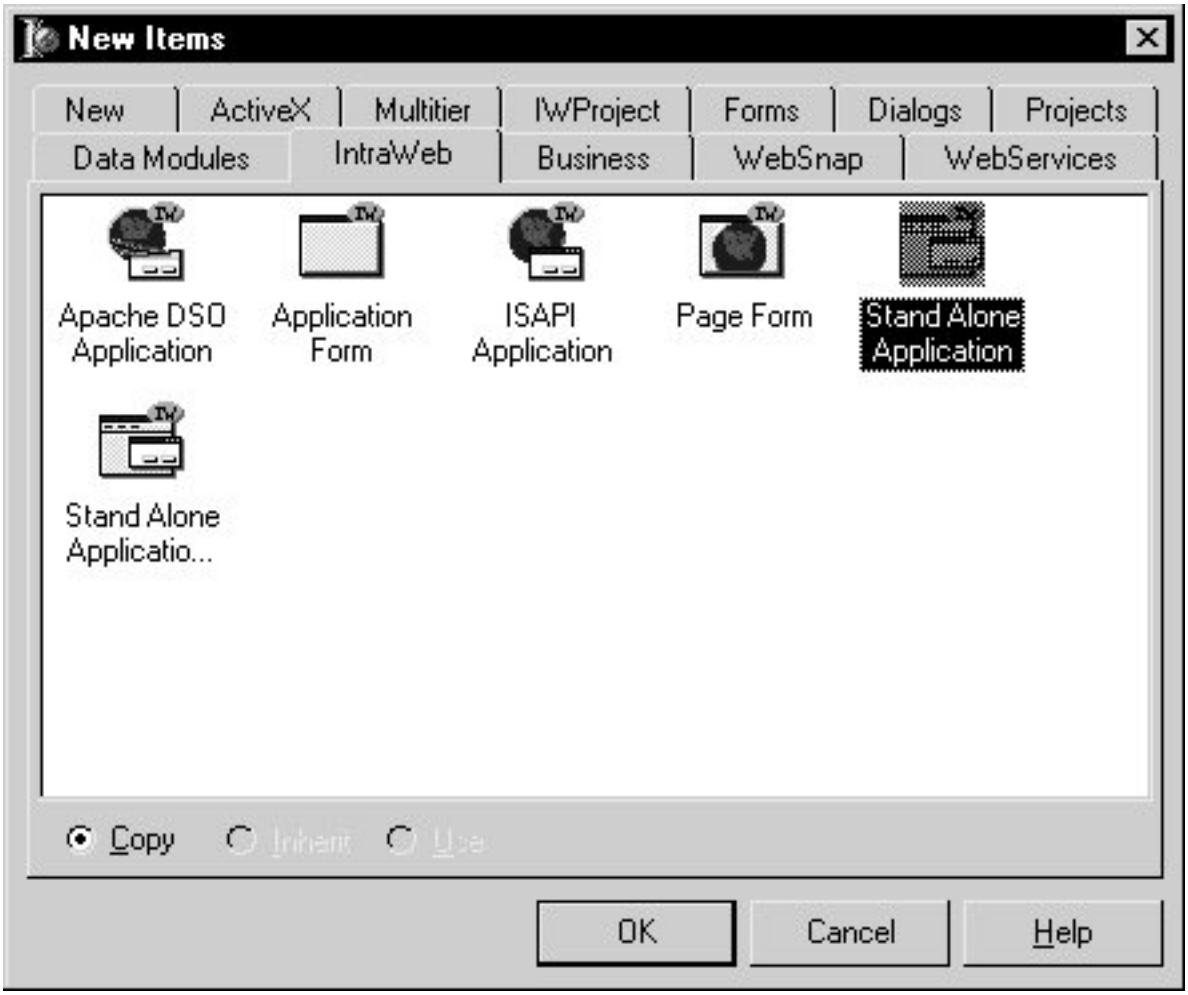
1. Creating a new IntraWeb application.
2. Editing the main form.
3. Writing an event handler for the button.
4. Running the completed application.

Creating a New IntraWeb Application

The first step in the process of creating the demo program is to create a new IntraWeb project. The project will be a stand alone application, but you can convert it to ISAPI/NSAPI or Apache later by changing two lines of code.

To create the new project:

1. Using an external tool (such as Microsoft Windows Explorer), create a directory named Hello in your Projects directory. This is where the project files will be stored. IntraWeb will set the new project's name to match that of the directory.
2. Choose **File** ▸ **New** ▸ **Other**, then select the IntraWeb folder under Delphi Projects. The New Items dialog box appears.



3 Select Stand Alone Application and click OK.

4 Find your new Hello directory in the dialog box. Double-click it, then click OK.

You have just created your IntraWeb application in the Hello directory. All of its source code files have already been saved. You are now ready to edit the main form to create the Web user interface for your application.

For information about editing the main form, see Editing the Main Form.

Editing the Main Form

You are now ready to edit the main form to create the Web user interface for your application.

For information on creating a new IntraWeb Application, see Creating a New IntraWeb Application.

To create the Web user interface for your application:

1 Choose **File** ▸ **Open**, then select IWUnit1.pas and click OK. The main form window (named formMain) should appear in the IDE.

2 Click on the main form window. In the **Object Inspector**, change the form's *Title* property to "What is your name? " This question will appear in the title bar of the Web browser when you run the application and view the page corresponding to the main form.

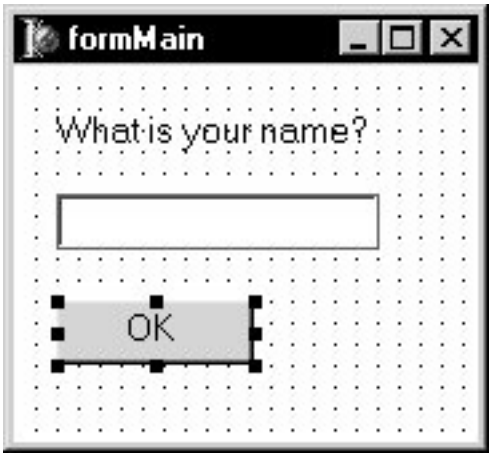
3 Drop an IWLabel component (found on the IW Standard tab of the **Tool palette**) onto the form. In the **Object Inspector**, change the *Caption* property to "What is your name?" That question should now appear on the form.

4 Drop an IWEdit component onto the form underneath the IWLabel component. Use the **Object Inspector** to make the following changes:

- Empty the contents of the Text property.
- Set the *Name* property to editName.

5 Drop an IWButton component on the form underneath the IWEdit component. Set its *Caption* property to OK.

Your form should look similar to this one:



You might want to save all your files before continuing.

For information about writing an event handler for the button, see Writing an Event Handler for the Button.

Writing an Event Handler for the Button

The form does not yet perform any actions when the user clicks the OK button.

For information on editing the main form, see Editing the Main Form.

You will now write an event handler that will display a greeting when the user clicks OK.

1 Double-click the OK button on the form. An empty event handler is created in the editor window, like the one shown here:

```
procedure TformMain.IWButton1Click(Sender: TObject);
begin

end;
```

2 Using the editor, add code to the event handler so it looks like the following:

```
procedure TformMain.IWButton1Click(Sender: TObject);
var s: string;
begin
   s := editName.Text;
   if Length(s) = 0 then
      WebApplication.ShowMessage("Please enter your name.")
   else
   begin
```

```
      WebApplication.ShowMessage("Hello, " + s +"!");
      editName.Text := "";
   end;
end;
```

For information about running the completed application, see Running the Completed Application.

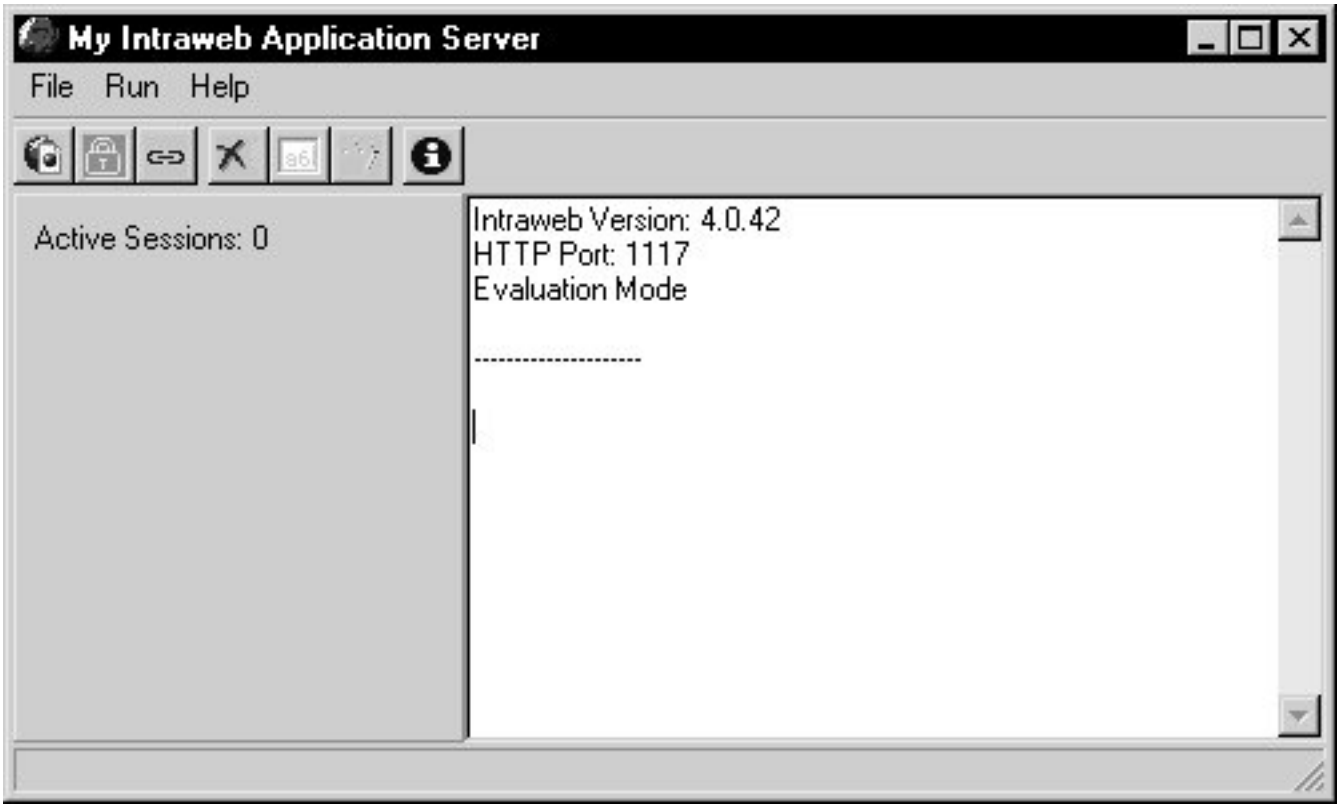
Running the Completed Application

You can now test the IntraWeb application.

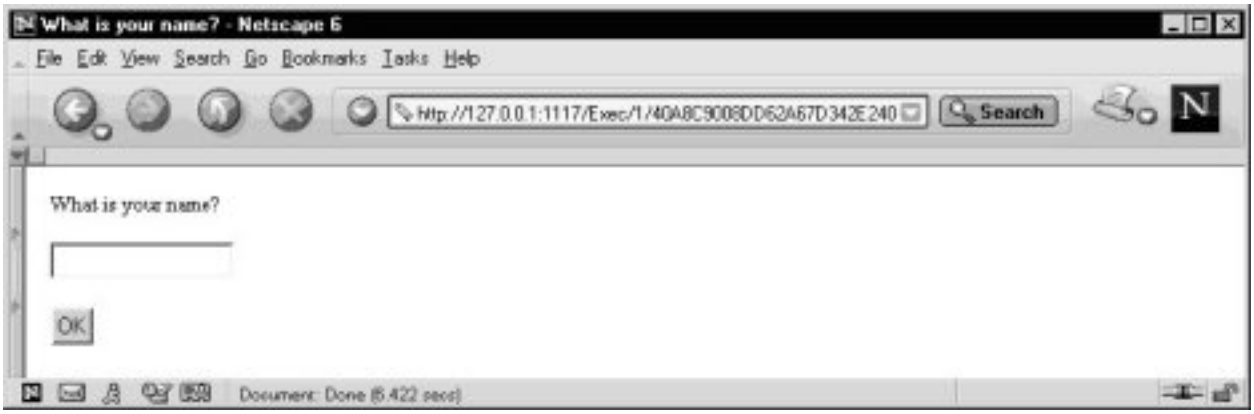
For information on writing the event handler, see Writing an Event Handler for the Button.

To test the IntraWeb application:

1 Select **Run** ▸ **Run**. The IntraWeb Application Server (shown below) will appear.



2 In the IntraWeb Application Server, select **Run** ▸ **Execute**. Your Web server application will appear in your default Web browser window. For example, here are the results in a Netscape 6 window:



3 Assume your name is World. Type World in the edit box and click the OK button. A modal dialog box will appear:

When you are finished using your application, you can terminate it by closing the browser window and then closing the IntraWeb Application Server.

Using IntraWeb with Web Broker and WebSnap

IntraWeb is a powerful tool for developing Web server applications all by itself. Still, there are some things it can't do alone, like create CGI applications. For CGI, you need Web Broker or WebSnap. Also, you may have existing Web Broker and WebSnap applications that you want to extend but not rewrite. You can still take advantage of IntraWeb's design tools by using IntraWeb forms and components in Web Broker or WebSnap projects. You can use IntraWeb to create individual pages instead of entire applications.

To create Web pages using IntraWeb tools, use the following steps:

1 Create or open a Web Broker or WebSnap application, and drop a WebDispatcher component on your Web module (Web Broker) or Web application module (WebSnap).

The WebDispatcher component is on the Internet tab of the **Tool palette**.

2 Drop an IWModuleController component on your Web module (Web Broker) or Web application module (WebSnap). IWModuleController is on the IW Control category of the **Tool palette**.

3 In WebSnap applications, create a new Web page module if necessary. In the New WebSnap Page dialog, uncheck the New File box in the HTML section before continuing.

Note: If you create a page module with the New File box checked, you can change the result later. Open the page module's unit file in the editor. Next, change '.html' to an empty string (") in the WebRequestHandler.AddWebModuleFactory call at the bottom of the unit.

4 Remove any existing page producer components from your Web module (Web Broker) or Web page module (WebSnap).

5 Drop an IWPageProducer component on your Web module or Web page module.

6 Select **File** ▸ **New** ▸ **Other** ▸ **IntraWeb** ▸ **Page Form** to create a new IntraWeb page form.

7 Add an *OnGetForm* event handler by double-clicking the IWPageProducer component on your Web module or Web page module. A new method will appear in the editor window.

8 Connect the IntraWeb form to the Web module or Web page module by adding a line of code to your *OnGetForm* event handler. The code line should be similar to, if not identical to, the following:

```
[Delphi]
VForm := TformMain.Create(AWebApplication);
```

```
[C++]
VForm = TformMain->Create(AWebApplication);
```

If necessary, change *TformMain* to the name of your IntraWeb form class. To find the form class name, click on the form. Its name appears next to the form window name in the **Object Inspector**.

9 In the unit file where you changed the event handler, add IWApplication and IWPageForm to the **uses** clause. Also, add the unit containing your form.

Working with XML documents

Working with XML Documents

XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on.

XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which describes a collection of stock holdings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

This example illustrates a number of typical elements in an XML document. The first line is a processing instruction called an XML declaration. The XML declaration is optional but you should include it, because it supplies useful information about the document. In this example, the XML declaration says that the document conforms to version 1.0 of the XML specification, that it uses UTF-8 character encoding, and that it relies on an external file for its document type declaration (DTD).

The second line, which begins with the <!DOCType> tag, is a document type declaration (DTD). The DTD is how XML defines the structure of the document. It imposes syntax rules on the elements (tags) contained in the document. The DTD in this example references another file (sth.dtd). In this case, the structure is defined in an external file, rather than in the XML document itself. Other types of files that describe the structure of an XML document include Reduced XML Data (XDR) and XML schemas (XSD).

The remaining lines are organized into a hierarchy with a single root node (the <StockHoldings> tag). Each node in this hierarchy contains either a set of child nodes, or a text value. Some of the tags (the <Stock> and <shares> tags) include attributes, which are Name=Value pairs that provide details on how to interpret the tag.

Although it is possible to work directly with the text in an XML document, typically applications use additional tools for parsing and editing the data. W3C defines a set of standard interfaces for representing a parsed XML document called the Document Object Model (DOM). A number of vendors provide XML parsers that implement the DOM interfaces to let you interpret and edit XML documents more easily.

Delphi provides a number of additional tools for working with XML documents. These tools use a DOM parser that is provided by another vendor, and make it even easier to work with XML documents.They include

- VCL components and interfaces for working with XML documents.
- An XML Data Binding wizard for generating classes to represent a particular XML document.
- Tools and components for converting between XML documents and data packets, which let you integrate XML documents into database applications.

Using the Document Object Model

The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. These interfaces are implemented by a number of different third-party vendors. If you do not want to use the default vendor that ships with Delphi, there is a registration mechanism that lets you integrate additional DOM implementations by other vendors into the XML framework.

The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces.

- To use one of the DOM vendors for which Delphi already includes support, locate the unit that represents the DOM implementation. These units end in the string 'xmldom.' For example, the unit for the Microsoft implementation is MSXMLDOM, the unit for the Xerces implementation is XERCESXMLDOM, and the unit for the Open XML implementation is OXMLDOM. If you add the unit for the desired implementation to your project, the DOM implementation is automatically registered so that it is available to your code.
- To use another DOM implementation, you must create a unit that defines a descendant of the TDOMVendor class. This unit can then work like one of the built-in DOM implementations, making your DOM implementation available when it is included in a project.
- In your descendant class, you must override two methods: the *Description* method, which returns a string identifying the vendor, and the *DOMImplementation* method, which returns the top-level interface (*IDOMImplementation*).
- Your new unit must register the vendor by calling the global RegisterDOMVendor procedure. This call typically goes in the initialization section of the unit.
- When your unit is unloaded, it needs to unregister itself to indicate that the DOM implementation is no longer available. Unregister the vendor by calling the global UnRegisterDOMVendor procedure. This call typically goes in the finalization section.

Some vendors supply extensions to the standard DOM interfaces. To allow you to uses these extensions, the XMLDOM unit also defines an *IDOMNodeEx* interface. *IDOMNodeEx* is a descendant of the standard *IDOMNode* that includes the most useful of these extensions.

You can work directly with the DOM interfaces to parse and edit XML documents. Simply call the GetDOM function to obtain an *IDOMImplementation* interface, which you can use as a starting point.

Note: For detailed descriptions of the DOM interfaces, see the declarations in the XMLDOM unit, the documentation supplied by your DOM Vendor, or the specifications provided on the W3C web site (www.w3.org).

You may find it more convenient to use special XML classes rather than working directly with the DOM interfaces. These are described in:

- Working with XML components
- Abstracting XML documents with the Data Binding wizard

Working with XML Components

The VCL defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents.

To use the XML classes for examining or editing an XML document you start by setting up an instance of TXMLDocument. You can then work with the nodes of the XML document component to examine or edit the body of the XML document.

Using TXMLDocument

The starting point for working with an XML document is the TXMLDocument component.

The following steps describe how to use TXMLDocument to work directly with an XML document:

1. Add a *TXMLDocument* component to your form or data module. *TXMLDocument* appears on the Internet category of the **Tool Palette**.
2. Set the DOMVendor property to specify the DOM implementation you want the component to use for parsing and editing an XML document. The **Object Inspector** lists all the currently registered DOM vendors. For information on DOM implementations, see Using the Document Object Model.
3. Depending on your implementation, you may want to set the ParseOptions property to configure how the underlying DOM implementation parses the XML document.
4. If you are working with an existing XML document, specify the document:
   - If the XML document is stored in a file, set the FileName property to the name of that file.
   - You can specify the XML document as a string instead by using the XML property.
5. Set the Active property to *True*.

Once you have an active *TXMLDocument* object, you can traverse the hierarchy of its nodes, reading or setting their values. The root node of this hierarchy is available as the DocumentElement property.

For information on working with the nodes of the XML document, see Working with XML nodes.

Working with XML Nodes

Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:

```
[Delphi]
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
```

```
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

TXMLDocument would generate a hierarchy of nodes as follows: The root of the hierarchy would be the *StockHoldings* node. *StockHoldings* would have two child nodes, which correspond to the two *Stock* tags. Each of these two child nodes would have four child nodes of its own (*name*, *price*, *symbol*, and *shares*). Those four child nodes would act as leaf nodes. The text they contain would appear as the value of each of the leaf nodes.

Note: This division into nodes differs slightly from the way a DOM implementation generates nodes for an XML document. In particular, a DOM parser treats all tagged elements as internal nodes. Additional nodes (of type text node) are created for the values of the *name*, *price*, *symbol*, and *shares* nodes. These text nodes then appear as the children of the *name*, *price*, *symbol*, and *shares* nodes.

Each node is accessed through an IXMLNode interface, starting with the root node, which is the value of the XML document component's DocumentElement property.

Working with a node's value

Given an *IXMLNode* interface, you can check whether it represents an internal node or a leaf node by checking the IsTextElement property.

- If it represents a leaf node, you can read or set its value using the Text property.
- If it represents an internal node, you can access its child nodes using the ChildNodes property.

Thus, for example, using the XML document above, you can read the price of Borland's stock as follows:

```
[Delphi]
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0];
Price := BorlandStock.ChildNodes['price'].Text;
```

```
[C++]
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0);
AnsiString Price = BorlandStock->ChildNodes->Nodes[WideString("price")]->Text;
```

Working with a node's attributes

If the node includes any attributes, you can work with them using the Attributes property. You can read or change an attribute value by specifying an existing attribute name. You can add new attributes by specifying a new attribute name when you set the *Attributes* property:

```
[Delphi]
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0];
BorlandStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

```
[C++]
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0);
BorlandStock->ChildNodes->Nodes[WideString("shares")]->Attributes[WideString("type")] =
"common";
```

Adding and deleting child nodes

You can add child nodes using the *AddChild* method. AddChild creates new nodes that correspond to tagged elements in the XML document. Such nodes are called element nodes.

To create a new element node, specify the name that appears in the new tag and, optionally, the position where the new node should appear. For example, the following code adds a new stock listing to the document above:

```
[Delphi]
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems'
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
```

```
[C++]
_di_IXMLNode NewStock = XMLDocument1->DocumentElement->AddChild(WideString("stock"));
NewStock->Attributes[WideString("exchange")] = "NASDAQ";
_di_IXMLNode ValueNode = NewStock->AddChild(WideString("name"));
ValueNode->Text = WideString("Cisco Systems");
ValueNode = NewStock->AddChild(WideString("price"));
ValueNode->Text = WideString("62.375");
ValueNode = NewStock->AddChild(WideString("symbol"));
ValueNode->Text = WideString("CSCO");
ValueNode = NewStock->AddChild(WideString("shares"));
ValueNode->Text = WideString("25");
```

An overloaded version of *AddChild* lets you specify the namespace URI in which the tag name is defined.

You can delete child nodes using the methods of the *ChildNodes* property. ChildNodes is an *IXMLNodeList* interface, which manages the children of a node. You can use its Delete method to delete a single child node that is identified by position or by name. For example, the following code deletes the last stock listed in the document above:

```
[Delphi]
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

```
[C++]
_di_IXMLNode StockList = XMLDocument1->DocumentElement;
StockList->ChildNodes->Delete(StockList->ChildNodes->Count - 1);
```

Abstracting XML Documents with the Data Binding Wizard

It is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*). However, you can write code that is much simpler and more readable by using the XML Data Binding wizard.

The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it. For example, given XML data that looks like the following:

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

The Data Binding wizard generates the following interface (along with a class to implement it):

```
[Delphi]
ICustomer = interface(IXMLNode)
        ['{8CD6A6E8-24FC-426F-9718-455F0C507C8E}']
        { Property Accessors }
        function Get_Id: Integer;
        function Get_Name: WideString;
        function Get_Phone: WideString;
        procedure Set_Id(Value: Integer);
        procedure Set_Name(Value: WideString);
        procedure Set_Phone(Value: WideString);
        { Methods & Properties }
        property Id: Integer read Get_Id write Set_Id;
        property Name: WideString read Get_Name write Set_Name;
        property Phone: WideString read Get_Phone write Set_Phone;
end;
```

```
[C++]
__interface INTERFACE_UUID("{F3729105-3DD0-1234-80e0-22A04FE7B451}") ICustomer :
    public IXMLNode
{
public:
  virtual int __fastcall Getid(void) = 0 ;
  virtual DOMString __fastcall Getname(void) = 0 ;
  virtual DOMString __fastcall Getphone(void) = 0 ;
  virtual void __fastcall Setid(int Value)= 0 ;
  virtual void __fastcall Setname(DOMString Value)= 0 ;
  virtual void __fastcall Setphone(DOMString Value)= 0 ;
  __property int id = {read=Getid, write=Setid};
  __property DOMString name = {read=Getname, write=Setname};
  __property DOMString phone = {read=Getphone, write=Setphone};
};
```

Every child node is mapped to a property whose name matches the tag name of the child node and whose value is the interface of the child node (if the child is an internal node) or the value of the child node (for leaf nodes). Every

node attribute is also mapped to a property, where the property name is the attribute name and the property value is the attribute value.

In addition to creating interfaces (and implementation classes) for each tagged element in the XML document, the wizard creates global functions for obtaining the interface to the root node. For example, if the XML above came from a document whose root node had the tag <Customers>, the Data Binding wizard would create the following global routines:

```
[Delphi]
function Getcustomers(Doc: IXMLDocument): IXMLCustomerType;
function Loadcustomers(const FileName: WideString): IXMLCustomerType;
function Newcustomers: IXMLCustomerType;
```

```
[C++]
extern PACKAGE _di_ICustomers __fastcall GetCustomers(TXMLDocument *XMLDoc);
extern PACKAGE _di_ICustomers __fastcall GetCustomers(_di_IXMLDocument XMLDoc);
extern PACKAGE _di_ICustomers __fastcall LoadCustomers(const WideString FileName);
extern PACKAGE _di_ICustomers __fastcall NewCustomers(void);
```

The Get... function takes the interface for a *TXMLDocument* instance . The Load... function dynamically creates a *TXMLDocument* instance and loads the specified XML file as its value before returning an interface pointer. The New... function creates a new (empty) *TXMLDocument* instance and returns the interface to the root node.

Using the generated interfaces simplifies your code, because they reflect the structure of the XML document more directly. For example, instead of writing code such as the following:

```
[Delphi]
CustIntf := XMLDocument1.DocumentElement;
CustName := CustIntf.ChildNodes[0].ChildNodes['name'].Value;
```

```
[C++]
_di_IXMLNode CustIntf = XMLDocument1->DocumentElement;
CustName = CustIntf->ChildNodes->Nodes->GetNode(0)->ChildNodes->Nodes[WideString("name")]-
>Value;
```

Your code would look as follows:

```
[Delphi]
CustIntf := GetCustomers(XMLDocument1);
CustName := CustIntf[0].Name;
```

```
[C++]
_di_ICustomers CustIntf = GetCustomers(XMLDocument1);
CustName = CustIntf->Nodes->GetNode(0)->Name;
```

Note that the interfaces generated by the Data Binding wizard all descend from *IXMLNode*. This means you can still add and delete child nodes in the same way as when you do not use the Data Binding wizard. (See the Adding and deleting child nodes section of Working with XML Nodes.) In addition, when child nodes represent repeating elements (when all of the children of a node are of the same type), the parent node is given two methods, *Add*, and *Insert*, for adding additional repeats. These methods are simpler than using *AddChild*, because you do not need to specify the type of node to create.

The following topics provide detailed information on using the XML Data Binding wizard:

- Using the XML Data Binding wizard

- Using code that the XML Data Binding wizard generates

Using the XML Data Binding Wizard

To use the Data Binding wizard:

1. Choose **File** ▸ **New** ▸ **Other** and select the icon labeled XML Data Binding from the right pane of the New folder located under Delphi Projects.
2. The XML Data Binding wizard appears.
3. On the first page of the wizard, specify the XML document or schema for which you want to generate interfaces. This can be a sample XML document, a Document Type Definition (.dtd) file, a Reduced XML Data (.xdr) file, or an XML schema (.xsd) file.
4. Click the Options button to specify the naming strategies you want the wizard to use when generating interfaces and implementation classes and the default mapping of types defined in the schema to native Delphi data types.
5. Move to the second page of the wizard. This page lets you provide detailed information about every node type in the document or schema. At the left is a tree view that shows all of the node types in the document. For complex nodes (nodes that have children), the tree view can be expanded to display the child elements. When you select a node in this tree view, the right-hand side of the dialog displays information about that node and lets you specify how you want the wizard to treat that node.

   - The Source Name control displays the name of the node type in the XML schema.
   - The Source Datatype control displays the type of the node's value, as specified in the XML schema.
   - The Documentation control lets you add comments to the schema describing the use or purpose of the node.
   - If the wizard generates code for the selected node (that is, if it is a complex type for which the wizard generates an interface and implementation class, or if it is one of the child elements of a complex type for which the wizard generates a property on the complex type's interface), you can use the Generate Binding check box to specify whether you want the wizard to generate code for the node. If you uncheck Generate Binding, the wizard does not generate the interface or implementation class for a complex type, or does not create a property in the parent interface for a child element or attribute.
   - The Binding Options section lets you influence the code that the wizard generates for the selected element. For any node, you can specify the Identifier Name (the name of the generated interface or property). In addition, for interfaces, you must indicate which one represents the root node of the document. For nodes that represent properties, you can specify the type of the property and, if the property is not an interface, whether it is a read-only property.

6. Once you have specified what code you want the wizard to generate for each node, move to the third page. This page lets you choose some global options about how the wizard generates its code and lets you preview the code that will be generated, and lets you tell the wizard how to save your choices for future use.

   - To preview the code the wizard generates, select an interface in the Binding Summary list and view the resulting interface definition in the Code Preview control.
   - Use the Data Binding Settings to indicate how the wizard should save your choices. You can store the settings as annotations in a schema file that is associated with the document (the schema file specified on the first page of the dialog), or you can name an independent schema file that is used only by the wizard.

7. When you click Finish, the Data Binding wizard generates a new unit that defines interfaces and implementation classes for all of the node types in your XML document. In addition, it creates a global function that takes a *TXMLDocument* object and returns the interface for the root node of the data hierarchy.

Using Code That the XML Data Binding Wizard Generates

Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components, your starting point is the TXMLDocument component that appears on the Internet category of the **Tool Palette**.

To work with an XML document, use the following steps:

1 Obtain an interface for the root node of your XML document. You can do this in one of three ways:

Method	Description

Place a *TXMLDocument* component in your form or data module. Bind the *TXMLDocument* to an XML document by setting the FileName property.	(As an alternative approach, you can use a string of XML by setting the XML property at runtime.) Then, In your code, call the global function that the wizard created to obtain an interface for the root node of the XML document. For example, if the root element of the XML document was the tag <StockList>, by default, the wizard generates a function *Getstocklist*, which returns an *IXMLStockListType*. interface:
	`[Delphi]`
	`var`
	`  StockList: IXMLStockListType;`
	`begin`
	`  XMLDocument1.FileName := 'Stocks.xml';`
	`  StockList := Getstocklist (XMLDocument1);`
	
	`[C++]`
	`XMLDocument1->FileName := "Stocks.xml";`
	`_di_IStockListType StockList = GetStockListType(XMLDocument1);`

Call the generated Load... function	Call the generated Load... function to create and bind the *TXMLDocument* instance and obtain its interface all in one step. For example, using the same XML document described above:
	`[Delphi]`
	`var`
	`  StockList: IXMLStockListType;`
	`begin`
	`  StockList := Loadstocklist ('Stocks.xml');`
	
	`[C++]`
	`_di_IStockListType StockList = LoadStockListType("Stocks.xml");`

Call the generated New... function	Call the generated New... function to create the *TXMLDocument* instance for an empty document when you want to create all the data in your application:

```
[Delphi]
    var
      StockList: IXMLStockListType;
    begin
      StockList := Newstocklist;
```

```
[C++]
    _di_IStockListType StockList =
NewStockListType();
```

---

2 This interface has properties that correspond to the subnodes of the document's root element, as well as properties that correspond to that root element's attributes. You can use these to traverse the hierarchy of the XML document, modify the data in the document, and so on.

3 To save any changes you make using the interfaces generated by the wizard, call the *TXMLDocument* component's SaveToFile method or read its XML property.

Tip: If you set the *Options* property of the *TXMLDocument* object to include *doAutoSave*, then you do not need to explicitly call the *SaveToFile* method.

Using Web Services

Using Web Services

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at

```
http://www.w3.org/TR/SOAP/
```

Note: Although the components that support Web Services are built to use SOAP and HTTP, the framework is sufficiently general that it can be expanded to use other encoding and communications protocols.

In addition to letting you build SOAP-based Web Service applications (servers), special components and wizards let you build clients of Web Services that use either a SOAP encoding or a Document Literal style. The Document Literal style is used in .Net Web Services.

Web Service applications publish information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code as well when importing the WSDL document.

The following topics describe support for working with Web Services in greater detail:

- Understanding invokable interfaces
- Writing servers that support Web Services
- Writing clients for Web Services

Understanding Invokable Interfaces

Servers that support Web Services are built using invokable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). On the server, this RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled. On clients, this RTTI is used to dynamically generate a method table for making calls to the methods of the interface.

To create an invokable interface, you need only compile an interface with the {$M+} compiler option. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, your Web Service can only use the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be used as part of the Web Service.

When defining a Web service, you can derive an invokable interface from the base invokable interface, IInvokable. *IInvokable* is defined in the System unit. *IInvokable* is the same as the base interface (*IInterface*), except that it is compiled using the {$M+} compiler option. The {$M+} compiler option ensures that the interface and all its descendants include RTTI.

For example, the following code defines an invokable interface that contains two methods for encoding and decoding numeric values:

```
[Delphi]
IEncodeDecode = interface(IInvokable)
['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

```
[C++]
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IEncodeDecode :
    public IInvokable
{
public:
  virtual double __stdcall EncodeValue(int Value) = 0 ;
  virtual int __stdcall DecodeValue(double Value) = 0 ;
};
```

Note: An invokable interface can use overloaded methods, but only if the different overloads can be distinguished by parameter count. That is, one overload must not have the same number of parameters as another, including the possible number of parameters when default parameters are taken into account.

Before a Web Service application can use this invokable interface, it must be registered with the invocation registry. On the server, the invocation registry entry allows the invoker component (THTTPSOAPPascalInvoker) to identify an implementation class to use for executing interface calls. On client applications, an invocation registry entry allows remote interfaced objects (THTTPRio) to look up information that identifies the invokable interface and supplies information on how to call it.

Typically, your Web Service client or server creates the code to define invokable interfaces either by importing a WSDL document or using the Web Service wizard. By default, when the WSDL importer or Web Service wizard generates an interface, the definition is added to a unit with the same name as the Web Service. This unit includes both the interface definition and code to register the interface with the invocation registry. The invocation registry is a catalog of all registered invokable interfaces, their implementation classes, and any functions that create instances of the implementation classes. It is accessed using the global *InvRegistry* function, which is defined in the InvokeRegistry unit.

The definition of the invokable interface is added to the interface section of the unit, and the code to register the interface goes in the initialization section. The registration code looks like the following:

```
[Delphi]
initialization
  InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

```
[C++]
static void RegTypes()
{
  InvRegistry()->RegisterInterface(__delphirtti(IEncodeDecode), "", "");
}
#pragma startup RegTypes 32
```

Note: The implementation section's uses clause must include the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

The interfaces of Web Services must have a namespace to identify them among all the interfaces in all possible Web Services. The previous example does not supply a namespace for the interface. When you do not explicitly supply a namespace, the invocation registry automatically generates one for you. This namespace is built from a string that uniquely identifies the application (the *AppNamespacePrefix* variable), the interface name, and the name of the unit in which it is defined. If you do not want to use the automatically-generated namespace, you can specify one explicitly using a second parameter to the *RegisterInterface* call.

You can use the same unit file to define an invokable interface for both client and server applications. If you are doing this, it is a good idea to keep the unit that defines your invokable interfaces separate from the unit in which you write the classes that implement them. Because the generated namespace includes the name of the unit in which the interface is defined, sharing the same unit in both client and server applications enables them to automatically use the same namespace, as long as they both use the same value for the *AppNamespacePrefix* variable.

The preceding example uses only scalar types (integers and doubles) in the methods of the interface. You can use nonscalar types as well, but they require a bit more work.

Using Nonscalar Types in Invokable Interfaces

The Web Services architecture automatically includes support for marshaling the following scalar types:

Boolean	ByteBool	WordBool
LongBool	Char	Byte
ShortInt	SmallInt	Word
Integer	Cardinal	LongInt
Int64	Single	Double
Extended	string	WideString
Currency	TDateTime	Variant

You need do nothing special when you use these scalar types on an invokable interface. If your interface includes any properties or methods that use other types, however, your application must register those types with the remotable type registry.

Dynamic arrays can be used in invokable interfaces. They must be registered with the remotable type registry, but this registration happens automatically when you register the interface. The remotable type registry extracts all the information it needs from the type information that the compiler generates.

Note: You should avoid defining multiple dynamic array types with the same element type. Because the compiler treats these as transparent types that can be implicitly cast one to another, it doesn't distinguish their runtime type information. As a result, the remotable type registry can't distinguish the types. This is not a problem for servers, but can result in clients using the wrong type definition. As an alternate approach, you can use remotable clases to represent array types.

Note: The dynamic array types defined in the Types unit are automatically registered for you, so your application does not need to add any special registration code for them. One of these in particular, *TByteDynArray*, deserves special notice because it maps to a 'base64' block of binary data, rather than mapping each array element separately the way the other dynamic array types do.

Enumerated types and types that map directly to one of the automatically-marshaled scalar types can also be used in an invokable interface. As with dynamic array types, they are automatically registered with the remotable type registry.

For any other types, such as static arrays, structs or records, sets, interfaces, or classes, you must map the type to a remotable class. A remotable class is a class that includes runtime type information (RTTI). Your interface must then use the remotable class instead of the corresponding static array, struct or record, set, interface, or class. Any remotable classes you create must be registered with the remotable type registry. As with other types, this registration happens automatically.

Registering Nonscalar Types

Before an invokable interface can use any types other than the built-in scalar types listed in Using nonscalar types in invokable interfaces, the application must register the type with the remotable type registry. To access the remotable type registry, you must add the InvokeRegistry unit to your uses clause. This unit declares a global function, *RemTypeRegistry*, which returns a reference to the remotable type registry.

Note: On clients, the code to register types with the remotable type registry is generated automatically when you import a WSDL document. For servers, remotable types are registered for you automatically when you register an interface that uses them. You only need to explicitly add code to register types if you want to specify the namespace or type name rather than using the automatically-generated values.

The remotable type registry has two methods that you can use to register types: RegisterXSInfo and RegisterXSClass. The first (*RegisterXSInfo*) lets you register a dynamic array or other type definition. The second (*RegisterXSClass*) is for registering remotable classes that you define to represent other types.

If you are using dynamic arrays or enumerated types, the invocation registry can get the information it needs from the compiler-generated type information. Thus, for example, your interface may use a type such as the following:

```
[Delphi]
type
  TDateTimeArray = array of TXSDateTime;
```

```
[C++]
typedef DynamicArray<TXSDateTime> TDateTimeArray;
```

This type is registered automatically when you register the invokable interface. However, if you want to specify the namespace in which the type is defined or the name of the type, you must add code to explicitly register the type using the *RegisterXSInfo* method of the remotable type registry.

The registration goes in the initialization section of the unit where you declare or use the dynamic array:

```
[Delphi]
RemTypeRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, 'DTarray',
'DTarray');
```

```
[C++]
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(__arraytypeinfo(TDateTimeArray),
        MyNameSpace, "DTarray", "DTarray");
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter of *RegisterXSInfo* is the type information for the type you are registering. The second parameter is the namespace URI for the namespace in which the type is defined. If you omit this parameter or supply an empty string, the registry generates a namespace for you. The third parameter is the name of the type as it appears in native code. If you omit this parameter or supply an empty string, the registry uses the type name from the type information you supplied as the first parameter. The final parameter is the name of the type as it appears in WSDL documents. If you omit this parameter or supply an empty string, the registry uses the native type name (the third parameter).

Registering a remotable class is similar, except that you supply a class reference rather than a type information pointer. For example, the following line comes from the XSBuiltIns unit. It registers *TXSDateTime*, a *TRemotable* descendant that represents *TDateTime* values:

```
[Delphi]
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNameSpace, 'dateTime', '',True);
```

```
[C++]
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TXSDateTime), XMLSchemaNameSpace,
"dateTime", "", true);
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter is class reference for the remotable class that represents the type. The second is a uniform resource identifier (URI) that uniquely identifies the namespace of the new class. If you supply an empty string, the registry generates a URI for you. The third and fourth parameters specify the native and external names of the data type your class represents. If you omit the fourth parameter, the type registry uses the third parameter for both values. If you supply an empty string for both parameters, the registry uses the class name. The fifth parameter indicates whether the value of class instances can be transmitted as a string. You can optionally add a sixth parameter (not shown here) to control how multiple references to the same object instance should be represented in SOAP packets.

Using Remotable Objects

Use TRemotable as a base class when defining a class to represent a complex data type on an invokable interface. For example, in the case where you would ordinarily pass a record or struct as a parameter, you would instead define a *TRemotable* descendant where every member of the record or struct is a published property on your new class.

You can control whether the published properties of your *TRemotable* descendant appear as element nodes or attributes in the corresponding SOAP encoding of the type. To make the property an attribute, use the stored directive on the property definition, assigning a value of AS_ATTRIBUTE:

```
[Delphi]
property MyAttribute: Boolean read FMyAttribute write FMyAttribute stored AS_ATTRIBUTE;
```

```
[C++]
__property bool MyAttribute = {read=FMyAttribute, write=FMyAttribute, stored= AS_ATTRIBUTE;
```

Note: If you do not include a stored directive, or if you assign any other value to the stored directive (even a function that returns AS_ATTRIBUTE), the property is encoded as a node rather than an attribute.

If the value of your new *TRemotable* descendant represents a scalar type in a WSDL document, you should use TRemotableXS as a base class instead. *TRemotableXS* is a *TRemotable* descendant that introduces two methods for converting between your new class and its string representation. Implement these methods by overriding the XSToNative and NativeToXS methods.

For certain commonly-used XML scalar types, the XSBuiltIns unit already defines and registers remotable classes for you. These are listed in the following table:

Remotable classes

XML type	remotable class
dateTime timeInstant	TXSDateTime
date	TXSDate
time	TXSTime
durationtimeDuration	TXSDuration
decimal	TXSDecimal
hexBinary	TXSHexBinary

After you define a remotable class, it must be registered with the remotable type registry, as described in Registering nonscalar types. This registration happens automatically on servers when you register the interface that uses the class. On clients, the code to register the class is generated automatically when you import the WSDL document that defines the type. For an example of defining and registering a remotable class, see Remotable object example.

Tip: It is a good idea to implement and register *TRemotable* descendants in a separate unit from the rest of your server application, including from the units that declare and register invokable interfaces. In this way, you can use the type for more than one interface.

Representing attachments

One important *TRemotable* descendant is TSoapAttachment. This class represents an attachment. It can be used as the value of a parameter or the return value of a method on an invokable interface. Attachments are sent with SOAP messages as separate parts in a multipart form.

When a Web Service application or the client of a Web Service receives an attachment, it writes the attachment to a temporary file. *TSoapAttachment* lets you access that temporary file or save its content to a permanent file or stream. When the application needs to send an attachment, it creates an instance of *TSoapAttachment* and assigns its content by specifying the name of a file, supplying a stream from which to read the attachment, or providing a string that represents the content of the attachment.

Managing the lifetime of remotable objects

One issue that arises when using *TRemotable* descendants is the question of when they are created and destroyed. Obviously, the server application must create its own local instance of these objects, because the caller's instance

is in a separate process space. To handle this, Web Service applications create a data context for incoming requests. The data context persists while the server handles the request, and is freed after any output parameters are marshaled into a return message. When the server creates local instances of remotable objects, it adds them to the data context, and those instances are then freed along with the data context.

In some cases, you may want to keep an instance of a remotable object from being freed after a method call. For example, if the object contains state information, it may be more efficient to have a single instance that is used for every message call. To prevent the remotable object from being freed along with the data context, change its DataContext property.

Remotable Object Example

This example shows how to create a remotable object for a parameter on an invokable interface where you would otherwise use an existing class. In this example, the existing class is a string list (*TStringList*). To keep the example small, it does not reproduce the *Objects* property of the string list.

Because the new class is not scalar, it descends from *TRemotable* rather than *TRemotableXS*. It includes a published property for every property of the string list you want to communicate between the client and server. Each of these remotable properties corresponds to a remotable type. In addition, the new remotable class includes methods to convert to and from a string list.

```
[Delphi]
TRemotableStringList = class(TRemotable)
  private
    FCaseSensitive: Boolean;
    FSorted: Boolean;
    FDuplicates: TDuplicates;
    FStrings: TStringDynArray;
  public
    procedure Assign(SourceList: TStringList);
    procedure AssignTo(DestList: TStringList);
  published
    property CaseSensitive: Boolean read FCaseSensitive write FCaseSensitive;
    property Sorted: Boolean read FSorted write FSorted;
    property Duplicates: TDuplicates read FDuplicates write FDuplicates;
    property Strings: TStringDynArray read FStrings write FStrings;
end;
```

```
[C++]
class TRemotableStringList: public TRemotable
{
  private:
    bool FCaseSensitive;
    bool FSorted;
    Classes::TDuplicates FDuplicates;
    System::TStringDynArray FStrings;
  public:
    void __fastcall Assign(Classes::TStringList *SourceList);
    void __fastcall AssignTo(Classes::TStringList *DestList);
__published:
    __property bool CaseSensitive = {read=FCaseSensitive, write=FCaseSensitive};
    __property bool Sorted = {read=FSorted, write=FSorted};
    __property Classes::TDuplicates Duplicates = {read=FDuplicates, write=FDuplicates};
    __property System::TStringDynArray Strings = {read=FStrings, write=FStrings};
}
```

Note that *TRemotableStringList* exists only as a transport class. Thus, although it has a *Sorted* property (to transport the value of a string list's *Sorted* property), it does not need to sort the strings it stores, it only needs to record whether

the strings should be sorted. This keeps the implementation very simple. You only need to implement the *Assign* and *AssignTo* methods, which convert to and from a string list:

```
[Delphi]
procedure TRemotableStringList.Assign(SourceList: TStrings);
var I: Integer;
begin
  SetLength(Strings, SourceList.Count);
  for I := 0 to SourceList.Count - 1 do
    Strings[I] := SourceList[I];
  CaseSensitive := SourceList.CaseSensitive;
  Sorted := SourceList.Sorted;
  Duplicates := SourceList.Duplicates;
end;
procedure TRemotableStringList.AssignTo(DestList: TStrings);
var I: Integer;
begin
  DestList.Clear;
  DestList.Capacity := Length(Strings);
  DestList.CaseSensitive := CaseSensitive;
  DestList.Sorted := Sorted;
  DestList.Duplicates := Duplicates;
  for I := 0 to Length(Strings) - 1 do
    DestList.Add(Strings[I]);
end;
```

```
[C++]
void __fastcall TRemotableStringList::Assign(Classes::TStringList *SourceList)
{
  SetLength(Strings, SourceList->Count);
  for (int i = 0; i < SourceList->Count; i++)
    Strings[i] = SourceList->Strings[i];
  CaseSensitive = SourceList->CaseSensitive;
  Sorted = SourceList->Sorted;
  Duplicates = SourceList->Duplicates;
}
void __fastcall TRemotableStringList::AssignTo(Classes::TStringList *DestList)
{
  DestList->Clear();
  DestList->Capacity = Length(Strings);
  DestList->CaseSensitive = CaseSensitive;
  DestList->Sorted = Sorted;
  DestList->Duplicates = Duplicates;
  for (int i = 0; i < Length(Strings); i++)
    DestList->Add(Strings[i]);
}
```

Optionally, you may want to register the new remotable class so that you can specify its class name. If you do not register the class, it is registered automatically when you register the interface that uses it. Similarly, if you register the class but not the *TDuplicates* and *TStringDynArray* types that it uses, they are registered automatically. This code shows how to register the *TRemotableStringList* class and the *TDuplicates* type. *TStringDynArray* is registered automatically because it is one of the built-in dynamic array types declared in the Types unit.

This registration code goes in the initialization section of the unit where you define the remotable class:

```
[Delphi]
RemClassRegistry.RegisterXSInfo(TypeInfo(TDuplicates), MyNameSpace, 'duplicateFlag');
RemClassRegistry.RegisterXSClass(TRemotableStringList, MyNameSpace, 'stringList',
'',False);
```

```
[C++]
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TRemotableStringList), MyNameSpace,
"stringList", "", false);
}
#pragma startup initServices 32
```

Writing Servers that Support Web Services

In addition to the invokable interfaces and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (THTTPSoapDispatcher) receives incoming SOAP messages and passes them on to the invoker. The invoker (THTTPSOAPPascalInvoker) interprets the SOAP message, identifies the invokable interface it calls, executes the call, and assembles the response message.

Note: *THTTPSoapDispatcher* andTHTTPSoapPascalInvoker are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.

Once you register your invokable interfaces and their implementation classes, the dispatcher and invoker automatically handle any messages that identify those interfaces in the SOAP Action header of the HTTP request message.

Web services also include a publisher (TWSDLHTMLPublish). Publishers respond to incoming client requests by creating the WSDL documents that describe how to call the Web Services in the application.

Building a Web Service server

Developer Studio 2006 provides a wizard to speed development of a Web Service server application.

Use the following steps to build a server application that implements a Web Service:

1. Choose **File** ▸ **New** ▸ **Other** and on the WebServices page, double-click the Soap Server Application icon to launch the SOAP Server Application wizard. The wizard creates a new Web server application that includes the components you need to respond to SOAP requests.
2. When you exit the SOAP Server Application wizard, it asks you if you want to define an interface for your Web Service.

   If you are creating a Web Service from scratch, click yes, and you will see the Add New Web Service wizard. The wizard adds code to declare and register a new invokable interface for your Web Service. Edit the generated code to define and implement your Web Service. If you want to add additional interfaces (or you want to define the interfaces at a later time), choose **File** ▸ **New** ▸ **Other**, and on the WebServices page, double-click the SOAP Web Service interface icon. For details on using the Add New Web Service wizard and completing the code it generates, see Adding new Web Services.
3. If you are implementing a Web Service that has already been defined in a WSDL document, you can use the WSDL importer to generate the interfaces, implementation classes, and registration code that your application needs. You need only fill in the body of the methods the importer generates for the implementation classes. For details on using the WSDL importer, see Using the WSDL importer.

4 If you want to use the headers in the SOAP envelope that encodes messages between your application and clients, you can define classes to represent those headers and write code to process them. This is described in Defining and using SOAP headers.

5 If your application raises an exception when attempting to execute a SOAP request, the exception will be automatically encoded in a SOAP fault packet, which is returned instead of the results of the method call. If you want to convey more information than a simple error message, you can create your own exception classes that are encoded and passed to the client. This is described in Creating custom exception classes for Web Services.

6 The SOAP Server Application wizard adds a publisher component (*TWSDLHTMLPublish)* to new Web Service applications. This enables your application to publish WSDL documents that describe your Web Service to clients. For information on the WSDL publisher, see Generating WSDL documents for a Web Service application.

Using the SOAP Application Wizard

Web Service applications are a special form of Web Server application. Because of this, support for Web Services is built on top of the Web Broker architecture. To understand the code that the SOAP Application wizard generates, therefore, it is helpful to understand the Web Broker architecture. Information about Web Server applications in general, and Web Broker in particular, can be found in Creating Internet server applications and Using Web Broker.

To launch the SOAP application wizard, choose **File** ▸ **New** ▸ **Other**, and on the WebServices page, double-click the Soap Server Application icon. Choose the type of Web server application you want to use for your Web Service. For information about different types of Web Server applications, see Types of Web server applications.

The wizard generates a new Web server application that includes a Web module which contains three components:

- An invoker component (THTTPSOAPPascalInvoker). The invoker converts between SOAP messages and the methods of any registered invokable interfaces in your Web Service application.
- A dispatcher component (THTTPSoapDispatcher). The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its *WebDispatch* property to identify the HTTP request messages to which your application responds. This involves setting the *PathInfo* property to indicate the path portion of any URL directed to your application, and the *MethodType* property to indicate the method header for request messages.
- A WSDL publisher (TWSDLHTMLPublish). The WSDL publisher publishes a WSDL document that describes your interfaces and how to call them. The WSDL document tells clients that how to call on your Web Service application. For details on using the WSDL publisher, see Generating WSDL documents for a Web Service application.

The SOAP dispatcher and WSDL publisher are auto-dispatching components. This means they automatically register themselves with the Web module so that it forwards any incoming requests addressed using the path information they specify in their *WebDispatch* properties. If you right-click on the Web module, you can see that in addition to these auto-dispatching components, it has a single Web action item named *DefaultHandler*.

DefaultHandler is the default action item. That is, if the Web module receives a request for which it can't find a handler (can't match the path information), it forwards that message to the default action item. *DefaultHandler* generates a Web page that describes your Web Service. To change the default action, edit this action item's *OnAction* event handler.

Adding New Web Services

To add a new Web Service interface to your server application, choose **File** ▸ **New** ▸ **Other**, and on the WebServices page double-click on the icon labeled SOAP Server Interface.

The Add New Web Service wizard lets you specify the name of the invokable interface you want to expose to clients, and generates the code to declare and register the interface and its implementation class. By default, the wizard

also generates comments that show sample methods and additional type definitions, to help you get started in editing the generated files.

Editing the generated code

The interface definitions appear in the interface section of the generated unit. This generated unit has the name you specified using the wizard. You will want to change the interface declaration, replacing the sample methods with the methods you are making available to clients.

The wizard generates an implementation class that descends from TInvokableClass and that supports the invokable interface). If you are defining an invokable interface from scratch, you must edit the declaration of the implementation class to match any edits you made to the generated invokable interface.

When adding methods to the invokable interface and implementation class, remember that the methods must only use remotable types. For information on remotable types and invokable interfaces, see Using nonscalar types in invokable interfaces.

Using a different base class

The Add New WebService wizard generates implementation classes that descend from *TInvokableClass*. This is the easiest way to create a new class to implement a Web Service. You can, however, replace this generated class with an implementation class that has a different base class (for example, you may want to use an existing class as a base class.) There are a number of considerations to take into account when you replace the generated implementation class:

- Your new implementation class must support the invokable interface directly. The invocation registry, with which you register invokable interfaces and their implementation classes, keeps track of what class implements each registered interface and makes it available to the invoker component when the invoker needs to call the interface. It can only detect that a class implements an interface if the interface is directly included in the class declaration. It does not detect support an interface if it is inherited along with a base class.
- Your new implementation class must include support for the *IInterface* methods that are part of any interface. This point may seem obvious, but it is an easy one to overlook.
- You must change the generated code that registers the implementation class to include a factory method to create instances of your implementation class.

This last point takes a bit of explanation. When the implementation class descends from *TInvokableClass* and does not replace the inherited constructor with a new constructor that includes one or more parameters, the invocation registry knows how to create instances of the class when it needs them. When you write an implementation class that does not descend from *TInvokableClass*, or when you change the constructor, you must tell the invocation registry how to obtain instances of your implementation class.

You can tell the invocation registry how to obtain instances of your implementation class by supplying it with a factory procedure. Even if you have an implementation class that descends from *TInvokableClass* and that uses the inherited constructor, you may want to supply a factory procedure anyway. For example, you can use a single global instance of your implementation class rather than requiring the invocation registry to create a new instance every time your application receives a call to the invokable interface.

The factory procedure must be of type *TCreateInstanceProc*. It returns an instance of your implementation class. If the procedure creates a new instance, the implementation object should free itself when the reference count on its interface drops to zero, as the invocation registry does not explicitly free object instances. The following code illustrates another approach, where the factory procedure returns a single global instance of the implementation class:

```
[Delphi]
procedure CreateEncodeDecode(out obj: TObject);
```

```
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {save a reference to the interface so that the global instance doesn't free itself }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { return global instance }
end;
```

```
[C++]
void __fastcall CreateEncodeDecode(System::TObject* &obj)
{
  if (!FEncodeDecode)
  {
    FEncodeDecode = new TEncodeDecodeImpl();
    // save a reference to the interface so that the global instance doesn't free itself
    TEncodeDecodeImpl->QueryInterface(FEncodeDecodeInterface);
  }
  obj = FEncodeDecode;
}
```

Note: In this example, *FEncodeDecodeInterface* is a variable of type *IEncodeDecode*.

You register the factory procedure with an implementation class by supplying it as a second parameter to the call that registers the class with the invocation registry. First, locate the call the wizard generated to register the implementation class. This appears in initialization section of the unit that defines the class. It looks something like the following:

```
[Delphi]
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

```
[C++]
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl));
```

Add a second parameter to this call that specifies the factory procedure:

```
[Delphi]
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

```
[C++]
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl), &CreateEncodeDecode);
```

Using the WSDL Importer

To use the WSDL importer, choose File|New|Other, and on the WebServices page double-click the icon labeled WSDL importer. In the dialog that appears, specify the file name of a WSDL document (or XML file) or provide the URL where that document is published.

Note: If you do not know the URL for the WSDL document you want to import, you can browse for one by clicking the button labeled Search UDDI. This launches the UDDI browser.

Tip: An advantage of using the UDDI browser, even if you know the location of the WSDL document, is that when you locate the WSDL document using a UDDI description, client applications get fail-over support.

If the WSDL document is on a server that requires authentication (or must be reached using a proxy server that requires authentication), you need to provide a user name and password before the wizard can retrieve the WSDL document. To supply this information, click the Options button and provide the appropriate connection information.

When you click the Next button, the WSDL importer displays the code it generates for every definition in the WSDL document that is compatible with the Web Services framework. That is, it only uses those port types that have a SOAP binding. You can configure the way the importer generates code by clicking the Options button and choosing the options you want.

You can use the WSDL importer when writing either a server or a client application. When writing a server, click the Options button and in the resulting dialog, check the option that tells the importer to generate server code. When you select this option, the importer generates implementation classes for the invokable interfaces, and you need only fill in the bodies of the methods.

Warning: If you import a WSDL document to create a server that implements a Web Service that is already defined, you must still publish your own WSDL document for that service. There may be minor differences in the imported WSDL document and the generated implementation. For example, if the WSDL document or XML schema file uses identifiers that are also keywords, the importer automatically adjusts their names so that the generated code can compile.

When you click Finish, the importer creates new units that define and register invokable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

As an alternate approach, you can use the command line WSDL importer instead. For a server, call the command line importer with the -Os option, as follows:

```
[Delphi]
WSDLIMP -Os -P -V MyWSDLDoc.wsdl
```

```
[C++]
WSDLIMP -Os -C -V MyWSDLDoc.wsdl
```

For a client application, call the command line importer without the -Os option:

```
[Delphi]
WSDLIMP -P -V MyWSDLDoc.wsdl
```

```
[C++]
WSDLIMP -C -V MyWSDLDoc.wsdl
```

Tip: The command line interpreter includes some options that are not available when you use the WSDL importer in the IDE. For details, see the help for WSDLIMP.

Browsing for Business Services

You can use the UDDI browser to locate and import the WSDL document that describes a Web Service. Launch the UDDI browser by clicking the UDDI button on the WSDL importer.

One of the advantages of using the UDDI browser is that client applications gain fail-over support. That is, if a request to the server returns a status code of 404, 405, or 410 (indicating that the requested interface or method is not

available), the client application automatically returns to the UDDI entry where you found the WSDL document and checks whether it has changed.

Understanding UDDI

UDDI stands for Universal Description, Discovery, and Integration. It is a generic format for registering services available through the Web. A number of public registries exist, which make information about registered services available. Ideally, these public registries all contain the same information, although there may be minor discrepancies due to differences in when they update their information.

UDDI registries contain information about more than just Web Services. The format is sufficiently general that it can be used to describe any business service. Entries in the UDDI registry are organized hierarchically; first by business, then by type of service, and lastly by detailed information within a service. This detailed information is called a *TModel*. A Web Service, which can include one or more invokable interfaces, makes up a single *TModel*. Thus, a single business service can include multiple Web Services, as well as other business information. Each *TModel* can include a variety of information, including contact information for people within the business, a description of the service, and technical details such as a WSDL document.

For example, consider a hypothetical business, Widgets Inc. This business might have two services, widget manufacturing and custom widget design. Under the widget manufacturing service, you might find two *TModels*, one for selling parts to Widgets Inc, and one for ordering widgets. Each of these could be a Web Service. Under the custom widget design service, you might find a Web Service for obtaining cost estimates, and another *TModel* that is not a Web Service, which gives the address of a Web site for viewing past custom designs.

Using the UDDI browser

The first step after you launch the UDDI browser from the WSDL importer is to indicate the UDDI registry you want to search. The public registries should all contain the same information, but there can be differences. In addition, you may be using an internal, private registry. Select a public registry from the drop-down in the upper left corner, or type in the address of a private registry you want to use.

The next step is to locate the business from which you want to import a Web Service. Enter the name of the business in the edit control labeled Name. Other controls let you specify whether the browser must match this name exactly, or whether you want a case-insensitive search or want to allow a partial match. You can also specify how many matches you want to fetch (if multiple businesses meet your criteria) and how to sort the results.

Once you have specified the search criteria, click the Find button to locate the business. All of the matches appear in the tree view in the upper right corner. Use this tree view to drill down, locating the service you want, and the *TModel* within that service that corresponds to the Web Service you want to import. As you select items in this tree view, the lower right portion of the browser provides information about the selected item. When you select a *TModel* that represents a Web Service with a WSDL document, the Import button becomes enabled. When you locate the Web Service you want to import, click the Import button.

Defining and Using SOAP Headers

The SOAP encoding of a request to your Web Service application and of the response your application sends include a set of header nodes. Some of these, such as the SOAP Action header, are generated and interpreted automatically. However, you can also define your own headers to customize the communication between your server and its clients. Typically, these headers contain information that is associated with the entire invokable interface, or even with the entire application, rather than just the method that is the subject of a single message.

Defining header classes

For each header you want to define, create a descendant of ISOAPHeaders. *TSOAPHeader* is a descendant of *TRemotable*. That is, SOAP header objects are simply special types of remotable objects. As with any remotable

object, you can add published properties to your *TSOAPHeader* descendant to represent the information that your header communicates. Once you have defined a SOAP header class, it must be registered with the remotable type registry. Note that unlike other remotable classes, which are registered automatically when you register an invokable interface that uses them, you must explicitly write code to register your header types.

TSOAPHeader defines two properties that are used to represent attributes of the SOAP header node. These are *MustUnderstand* and *Actor*. When the *MustUnderstand* attribute is *True*, the recipient of a message that includes the header is required to recognize it. If the recipient can't interpret a header with the *MustUnderstand* attribute, it must abort the interpretation of the entire message. An application can safely ignore any headers it does not recognize if their *MustUnderstand* attribute is not set. The use of *MustUnderstand* is qualified by the *Actor* property. *Actor* is a URI that identifies the application to which the header is directed. Thus, for example, if your Web Service application forwards requests on to another service for further processing, some of the headers in client messages may be targeted at that other service. If such a header includes the *MustUnderstand* attribute, you should not abort the request even if your application can't understand the header. Your application is only concerned with those headers that give its URL as the *Actor*.

Sending and receiving headers

Once you have defined and registered header classes, they are available for your application to use. When your application receives a request, the headers on that message are automatically converted into the corresponding *TSOAPHeader* descendants that you have defined. Your application identifies the appropriate header class by matching the name of the header node against the type name you used when you registered the header class or against a name you supply by registering the header class with the invocation registry. Any headers for which the application can't find a match in the remotable type registry are ignored (or, if their *MustUnderstand* attribute is *True*, the application generates a SOAP fault).

You can access the headers your application receives using the ISOAPHeaders interface. There are two ways to obtain this interface: from an instance of *TInvokableClass* or, if you are implementing your invokable interface without using *TInvokableClass*, by calling the global GetSOAPHeaders function.

Use the *Get* method of *ISOAPHeaders* to access the headers by name. For example:

```
[Delphi]
TServiceImpl.GetQuote(Symbol: string): Double;
var
  Headers: ISOAPHeaders;
  H: TAuthHeader;
begin
  Headers := Self as ISOAPHeaders;
  Headers.Get(AuthHeader, TSOAPHeader(H)); { Retrieve the authentication header }
  try
    if H = nil then
      raise ERemotableException.Create("SOAP header for authentication required");
    { code here to check name and password }
  finally
    H.Free;
  end;
  { now that user is authenticated, look up and return quote }
end;
```

If you want to include any headers in the response your application generates to a request message, you can use the same interface. *ISOAPHeaders* defines a *Send* method to add headers to the outgoing response. Simply create an instance of each header class that corresponds to a header you want to send, set its properties, and call *Send*:

```
[Delphi]
TServiceImpl.GetQuote(Symbol: string): Double;
var
```

```
  Headers: ISOAPHeaders;
  H: TQuoteDelay;
  TXSDuration Delay;
begin
  Headers := Self as ISOAPHeaders;
  { code to lookup the quote and set the return value }
  { this code sets the Delay variable to the time delay on the quote }
  H := TQuoteDelay.Create;
  H.Delay := Delay;
  Headers.OwnsSentHeaders := True;
  Headers.Send(H);
end;
```

Handling scalar-type headers

Some Web Services define and use headers that are simple types (such as an integer or string) rather than a complex structure that corresponds to a remotable type. However, Delphi's support for SOAP headers requires that you use a *TSOAPHeader* descendant to represent header types. You can define header classes for simple types by treating the *TSOAPHeader* class as a holder class. That is, the *TSOAPHeader* descendant has a single published property, which is the type of the actual header. To signal that the SOAP representation does not need to include a node for the *TSOAPHeader* descendant, call the remotable type registry's RegisterSerializeOptions method (after registering the header type) and give your header type an option of *xoSimpleTypeWrapper*.

Communicating the structure of your headers to other applications

If your application defines headers, you need to allow its clients to access those definitions. If those clients are also written in Delphi, you can share the unit that defines and registers your header classes with the client application. However, you may want to let other clients know about the headers you use as well. To enable your application to export information about its header classes, you must register them with the invocation registry. Registering a header class also associates that class with a header name that is defined within a namespace.

Like the code that registers your invokable interface, the code to register a header class for export is added to the initialization section of the unit in which it is defined. Use the global *InvRegistry* function to obtain a reference to the invocation registry and call its *RegisterHeaderClass* method, indicating the interface with which the header is associated:

```
[Delphi]
initialization
  InvRegistry.RegisterInterface(TypeInfo(IMyWebService)); {register the interface}
  InvRegistry.RegisterHeaderClass(TypeInfo(IMyWebService), TMyHeaderClass); {and the header}
end.
```

```
[C++]
static void RegTypes()
{
  // register the invokable interface:
  InvRegistry()->RegisterInterface(__delphirtti(IMyService), "", "");
// register the header to be used with it:
  InvRegistry()->RegisterHeaderClass(__delphirtti(IMyService), __classid(TMyHeader));
}
#pragma startup RegTypes 32
```

You can limit the header to a subset of the methods on the interface by subsequent calls to the *RegisterHeaderMethod* method.

Note: The implementation section's uses clause must include the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

Once you have registered your header class with the invocation registry, its description is added to WSDL documents when you publish your Web Service.

Note: This registration of your header class with the invocation registry is in addition to the registration of that class with the remotable type registry.

Creating Custom Exception Classes for Web Services

When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception.

By default, the client application raises a generic exception of type ERemotableException with the information from the SOAP fault packet. You can transmit additional, application-specific information by deriving an ERemotableException descendant. The values of any published properties you add to the exception class are included in the SOAP fault packet so that the client can raise an equivalent exception.

To use an *ERemotableException* descendant, you must register it with the remotable type registry. Thus, in the unit that defines your *ERemotableException* descendant, you must add the InvokeRegistry unit to the uses clause and add a call to the RegisterXSClass method of the object that the global *RemTypeRegistry* function returns.

If the client also defines and registers your *ERemotableException* descendant, then when it receives the SOAP fault packet, it automatically raises an instance of the appropriate exception class, with all properties set to the values in the SOAP fault packet.

To allow clients to import information about your *ERemotableException* descendant, you must register it with the invocation registry as well as the remotable type registry. Add a call to the RegisterException method of the object that the global *InvRegistry* function returns.

Generating WSDL Documents for a Web Service Application

To allow client applications to know what Web Services your application makes available, you can publish a WSDL document that describes your invokable interfaces and indicates how to call them.

To publish a WSDL document that describes your Web Service, include a TWSDLHTMLPublish component in your Web Module. (The SOAP Server Application wizard adds this component by default.) *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the WebDispatch property to specify the path information of the URL that clients must use to access the list of WSDL documents. The Web browser can then request the list of WSDL documents by specifying an URL that is made up of the location of the server application followed by the path in the *WebDispatch* property. This URL looks something like the following:

```
http://www.myco.com/MyService.dll/WSDL
```

Tip: If you want to use a physical WSDL file instead, you can display the WSDL document in your Web browser and then save it to generate a WSDL document file.

Note: In addition to the WSDL document, the *THWSDLHTMLPublish* also generates a WS-Inspection document to describe the service for automated tools. The URL for this document looks something like the following:

```
http://www.myco.com/MyService.dll/inspection.wsil
```

It is not necessary to publish the WSDL document from the same application that implements your Web Service. To create an application that simply publishes the WSDL document, omit the code that implements and registers the implementation objects and only include the code that defines and registers invokable interfaces, remotable classes that represent complex types, and any remotable exceptions.

By default, when you publish a WSDL document, it indicates that the services are available at the same URL as the one where you published the WSDL document (but with a different path). If you are deploying multiple versions of your Web Service application, or if you are publishing the WSDL document from a different application than the one that implements the Web Service, you will need to change the WSDL document so that it includes updated information on where to locate the Web Service.

To change the URL, use the WSDL administrator. The first step is to enable the administrator. You do this by setting the AdminEnabled property of the *TWSDLHTMLPublish* component to true. Then, when you use your browser to display the list of WSDL documents, it includes a button to administer them as well. Use the WSDL administrator to specify the locations (URLs) where you have deployed your Web Service application.

Writing Clients for Web Services

You can write clients that access Web Services that you have written, or any other Web Service that is defined in a WSDL document. There are three steps to writing an application that is the client of a Web Service:

- Importing the definitions from a WSDL document.
- Obtaining an invokable interface and calling it to invoke the Web Service.
- Processing the headers of the SOAP messages that pass between the client and the server.

Importing WSDL Documents

Before you can use a Web Service, your application must define and register the invokable interfaces and types that are included in the Web Service application. To obtain these definitions, you can import a WSDL document (or XML file) that defines the service. The WSDL importer creates a unit that defines and registers the interfaces, headers, and types you need to use.

Calling Invokable Interfaces

To call an invokable interface, your client application must include any definitions of the invokable interfaces and any remotable classes that implement complex types.

If the server is written in Delphi, you can use the same units that the server application uses to define and register these interfaces and classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If it is automatically generated, the unit that defines the interfaces must have the same name in both client and server, and both client and server must define the global *AppNameSpacePrefix* variable to have the same value.

Once you have the definition of the invokable interface, there are two ways you can obtain an instance to call:

- If you imported a WSDL document, the importer automatically generates a global function that returns the interface, which you can then call.
- You can use a remote interfaced object.

Obtaining an invokable interface from the generated function

The WSDL importer automatically generates a function from which you can obtain the invokable interfaces you imported. For example, if you imported a WSDL document that defined an invokable interface named *IServerInterface*, the generated unit would include the following global function:

```
[Delphi]
function GetIServerInterface(UseWSDL: Boolean; Addr: string): IServerInterface;
```

```
[C++]
_di_IServerInterface GetIServerInterface(bool UseWSDL, AnsiString Addr);
```

The generated function takes two parameters: *UseWSDL* and *Addr*. *UseWSDL* indicates whether to look up the location of the server from a WSDL document (true), or whether the client application supplies the URL for the server (false).

When *UseWSDL* is false, *Addr* is the URL for the Web Service. When *UseWSDL* is true, *Addr* is the URL of a WSDL document that describes the Web Service you are calling. If you supply an empty string, this defaults to the document you imported. This second approach is best if you expect that the URL for the Web Service may change, or that details such as the namespace or SOAP Action header may change. Using this second approach, this information is looked up dynamically at the time your application makes the method call.

Note: The generated function uses an internal remote interfaced object to implement the invokable interface. If you are using this function and find you need to access that underlying remote interfaced object, you can obtain an *IRIOAccess* interface from the invokable interface, and use that to access the remote interfaced object:

```
[Delphi]
var
  Interf: IServerInterface;
  RIOAccess: IRIOAccess;
  X: THTTPRIO;
begin
  Intrf := GetIServerInterface(True,
          'http://MyServices.org/scripts/AppServer.dll/wsdl');
  RIOAccess := Intrf as IRIOAccess;
  X := RIOAccess.RIO as THTTPRIO;
```

Using a remote interfaced object

If you do not use the global function to obtain the invokable interface you want to call, you can create an instance of THTTPRio for the desired interface:

```
[Delphi]
X := THTTPRio.Create(nil);
```

```
[C++]
X = new THTTPRio(NULL);
```

Note: It is important that you do not explicitly destroy the *THTTPRio* instance. If it is created without an *Owner* (as in the previous line of code), it automatically frees itself when its interface is released. If it is created with an *Owner*, the *Owner* is responsible for freeing the *THTTPRio* instance.

Once you have an instance of *THTTPRio*, provide it with the information it needs to identify the server interface and locate the server. There are two ways to supply this information:

If you do not expect the URL for the Web Service or the namespaces and soap Action headers it requires to change, you can simply specify the URL for the Web Service you want to access. *THTTPRio* uses this URL to look up the definition of the interface, plus any namespace and header information, based on the information in the invocation registry. Specify the URL by setting the *URL* property to the location of the server:

```
[Delphi]
X.URL := 'http://www.myco.com/MyService.dll/SOAP/IServerInterface';
```

If you want to look up the URL, namespace, or Soap Action header from the WSDL document dynamically at runtime, you can use the *WSDLLocation*, *Service*, and *Port* properties, and it will extract the necessary information from the WSDL document:

```
[Delphi]
X.WSDLLocation := 'Cryptography.wsdl';
X.Service := 'Cryptography';
X.Port := 'SoapEncodeDecode';
```

After specifying how to locate the server and identify the interface, you can obtain an interface pointer for the invokable interface from the *THTTPRio* object. You obtain this interface pointer using the as operator. Simply cast the *THTTPRio* instance to the invokable interface:

```
[Delphi]
InterfaceVariable := X as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

```
[C++]
_di_IEncodeDecode InterfaceVariable;
if (X->QueryInterface(InterfaceVariable) == S_OK)
{
  Code = InterfaceVariable->EncodeValue(5);
}
```

When you obtain the interface pointer, *THTTPRio* creates a vtable for the associated interface dynamically in memory, enabling you to make interface calls.

THTTPRio relies on the invocation registry to obtain information about the invokable interface. If the client application does not have an invocation registry, or if the invokable interface is not registered, *THTTPRio* can't build its in-memory vtable.

Warning: If you assign the interface you obtain from *THTTPRio* to a global variable, you must change that assignment to nil before shutting down your application. For example, if *InterfaceVariable* in the previous code sample is a global variable, rather than stack variable, you must release the interface before the *THTTPRio* object is freed. Typically, this code goes in the *OnDestroy* event handler of the form or data module:

```
[Delphi]
procedure TForm1.FormDestroy(Sender: TObject);
begin
  InterfaceVariable := nil;
end;
```

```
[C++]
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  InterfaceVariable = NULL;
}
```

The reason you must reassign a global interface variable to nil is because *THTTPRio* builds its vtable dynamically in memory. That vtable must still be present when the interface is released. If you do not release the interface along with the form or data module, it is released when the global variable is freed on shutdown. The memory for global variables may be freed after the form or data module that contains the *THTTPRio* object, in which case the vtable will not be available when the interface is released.

Processing Headers in Client Applications

If the Web Service application you are calling expects your client to include any headers in its requests or if its response messages include special headers, your client application needs the definitions of the header classes that correspond to these headers. When you import a WSDL document that describes the Web Service application, the importer automatically generates code to declare these header classes and register them with the remotable type registry. If the server is written in Delphi, you can use the same units that the server application uses to define and register these header classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If it is automatically generated, the unit that defines the interfaces must have the same name in both client and server, and both client and server must define the global *AppSpacePrefix* variable to have the same value.

Note: For more information about header classes, see Defining and using SOAP headers.

As with a server, client applications use the ISOAPHeaders interface to access incoming headers and add outgoing headers. The remote interfaced object that you use to call invokable interfaces implements the *ISOAPHeaders* interface. However, you can't obtain an *ISOAPHeaders* interface directly from the remote interfaced object. This is because when you try to obtain an interface directly from a remote interfaced object, it generates an in-memory vtable, assuming that the interface is an invokable interface. Thus, you must obtain the *ISOAPHeaders* interface from the invokable interface rather than from the remote interfaced object:

```
var
  Service: IMyService;
  Hdr: TAuthHeader;
  Val: Double;
begin
  Service := HTTPRIO1 as IService;
  Hdr := TAUthHeader.Create;
  try
    Hdr.Name := "Frank Borland";
    Hdr.Password := "SuperDelphi";
    (Service as ISOAPHeaders).Send(Hdr); { add the header to outgoing message }
    Val := Service.GetQuote("BORL"); { invoke the service }
  finally
    Hdr.Free;
  end;
end;
```

Working with sockets

Working with Sockets

The socket components let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

To write applications that use sockets, you should understand

- Implementing services
- Types of socket connections
- Describing sockets
- Using socket components
- Responding to socket events
- Reading and writing over socket connections

Implementing Services

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

To implement or use a service using sockets, you must understand

- service protocols
- services and ports

Understanding Service Protocols

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the clients that use it. You can copy the API of a standard third party server (such as Apache), or you can design and publish your own API.

Services and Ports

Most standard services are associated, by convention, with specific port numbers. When implementing services, you can consider the port number a numeric code for the service.

Types of Socket Connections

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

Client Connections

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

Listening Connections

Server sockets do not locate clients. Instead, they form passive "half connections" that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as

they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

Server Connections

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

Describing Sockets

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies:

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its endpoints. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

Describing the Host

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.ASite.com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. You can learn the host name associated with any IP address (if one already exists) by executing the following command from a command prompt:

```
nslookup IPADDRESS
```

where IPADDRESS is the IP address you're interested in. If your local IP address doesn't have a host name and you decide you want one, contact your network administrator. It is common for computers to refer to themselves with the name *localhost* and the IP number 127.0.0.1.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

Using Ports

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

One way to look at port numbers is as numeric codes for the services implemented by network applications. This is a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

Using Socket Components

The Internet category includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are:

- TTcpServer
- TTcpClient
- TUdpSocket

Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.

Getting Information About the Connection

After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the LocalHost and LocalPort properties

to determine the address and port number used by the local client or server socket, or use the RemoteHost and RemotePort properties to determine the address and port number used by the remote client or server socket. Use the GetSocketAddr method to build a valid socket address based on the host name and port number. You can use the LookupPort method to look up the port number. Use the LookupProtocol method to look up the protocol number. Use the LookupHostName method to look up the host name based on the host machine's IP address.

To view network traffic in and out of the socket, use the BytesSent and BytesReceived properties.

Using Client Sockets

Add a TTcpClient or TUdpSocket component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client socket object to represent the client endpoint in a connection.

Use client sockets to

- Specify the desired server.
- Connect to the server.
- Get information about the connection.
- Read from or write to the server.
- Close the connection.

Specifying the Desired Server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the RemoteHost property to specify the remote host server by either its host name or IP address.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the RemotePort property to specify the server port number directly or indirectly by naming the target service.

Forming the Connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the Open method. If you want your application to form the connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.

Getting Information About the Connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the LocalHost and LocalPort properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the Handle property to obtain a handle to the socket connection to use when making socket calls.

Closing the Connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the Close method. The connection may also be closed from the server end. If that is the case, you will receive notification in an OnDisconnect event.

Using Server Sockets

Add a server socket component (TTcpServer or TUdpSocket) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that the server accepts.

Use server sockets to

- Specify the port.
- Listen for client requests.
- Connect to clients.
- Read from or write to the server.
- Close server connections.

Specifying the Port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the LocalPort property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

Listening for Client Requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the Open method. If you want your application to form the listening connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.

Connecting to Clients

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an OnAccept event.

Closing Server Connections

When you want to shut down the listening connection, call the Close method or set the Active property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When TCP clients shut down their individual connections to your server socket, you are informed by an OnDisconnect event.

Responding to Socket Events

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf*, *SendStream*, or *Sendln* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *Receiveln*. The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, you must use the *SendBuf* and *ReceiveBuf* methods to respond to these client events or server events.

Error Events

Client and server sockets generate OnError events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

Client Events

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An OnCreateHandle event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.
- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the OnConnect notification event occurs.

Server Events

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

Events when listening

Just before the listening connection is formed, the OnListening event occurs. You can use its Handle property to make changes to the socket before it is opened for listing. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListening* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An OnAccept event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If BlockMode is *bmThreadBlocking* an OnGetThread event occurs. If you want to provide your own customized descendant of ServerSocketThread, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerSocketThread*. If you want to perform any initialization of the thread, or make any socket API calls before the thread starts reading or writing over the connection, you should use the *OnGetThread* event handler for these tasks as well.
- The client completes the connection and an OnAccept event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

Reading and Writing Over Socket Connections

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

Non-blocking Connections

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection for client or server sockets, set the BlockMode property to *bmNonBlocking.*

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

Reading and Writing Events

Non-blocking sockets generate reading and writing events when they need to read or write over the connection. You can respond to these notifications in an *OnReceive* or *OnSend* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the ReceiveBuf or ReceiveIn method. To write to the socket connection, use the SendBuf, SendStream, or SendIn.

Blocking Connections

When the connection is blocking, your socket must initiate reading or writing over the connection. It cannot wait passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client or server sockets, set the BlockMode property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

Developing COM-based Applications

COM basics

Overview of COM Technologies

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM). For more information on clients, servers, and interfaces see Parts of a COM Application.

COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects adhere to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the written specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions designed for the purpose of creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly. You can find these wrappers defined in the `ComObj` unit and the API definitions in the `AxCtrls` unit.

Note: Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the `AxCtrls`, `OleCtrls`, and `OleServer` units. In addition, the Delphi interface to the COM API is in ActiveX.pas and ComSvcs.pas.

COM extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, and Active Directories. For details on COM extensions, see COM Extensions.

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see Implementing COM Objects with Wizards.

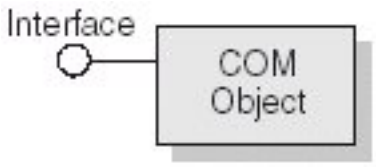
Parts of a COM Application

When implementing a COM application, you supply the following:

COM interface	The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods and properties. Note that COM properties are not identical to properties on VCL objects. COM properties always use read and write access methods.
COM server	A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.
COM client	The code that calls the interfaces to get the requested services from the server. Clients know what they want to get from the server (through the interface); clients do not know the internals of how the server provides the services. Delphi eases the process in creating a client by letting you install COM servers (such as a Word document or PowerPoint slide) as components on the **Tool Palette**. This allows you to connect to the server and hook its events through the **Object Inspector**.

COM Interfaces

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is as follows:



For example, every COM object must implement the basic interface, IUnknown. Through a routine called *QueryInterface* in *IUnknown*, clients can request other interfaces implemented by the server.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces are immutable; that is, they do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as *IMalloc* or *IPersist*.
- Interfaces are guaranteed to have a unique identification, called a **Globally Unique Identifier (GUID)**, which is a 128-bit randomly generated number. Interface GUIDs are called **Interface Identifiers (IIDs)**. This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer either explicitly or implicitly.

- Interfaces are not objects themselves; they provide a way to access an object. Therefore, clients do not access data directly; clients access data through an interface pointer. Windows 2000 adds an additional layer of indirection known as an interceptor through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the fundamental interface, IUnknown.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection. For more information, see In-process, out-of-process, and remote servers.

The Fundamental COM Interface, IUnknown

All COM objects must support the fundamental interface, called *IUnknown, a typedef* to the base interface type IInterface. *IUnknown* contains the following routines:

QueryInterface	Provides pointers to other interfaces that the object supports.
AddRef and Release	Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

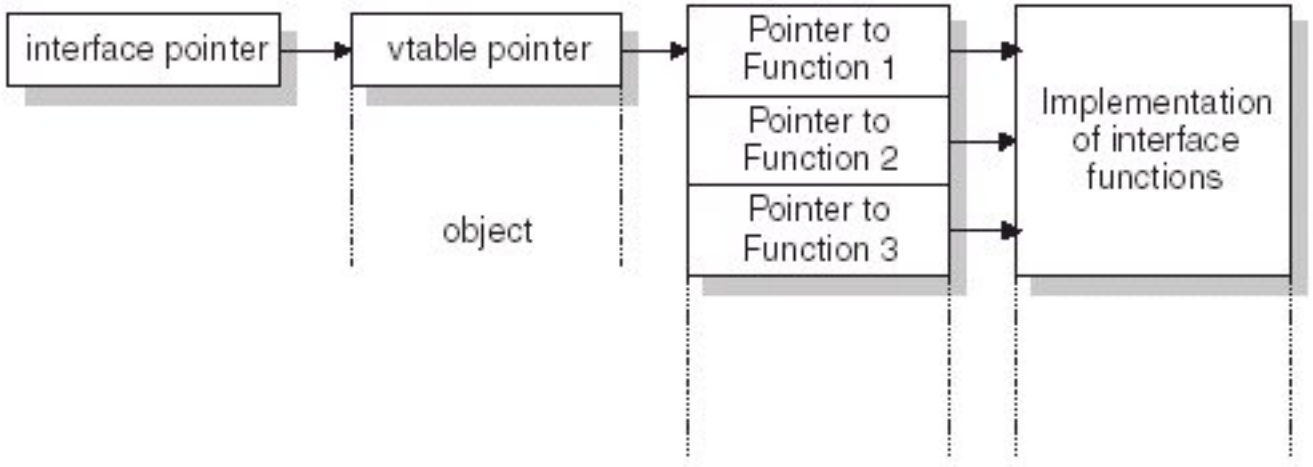
Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface*. *QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object(s).

COM Interface Pointers

An interface pointer is a pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in Delphi. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Delphi classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer *to the pointer* to the vtable, as shown in the following diagram.



In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, an added level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-in-time activation, whereby the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

COM Servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its interfaces.

In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's location.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server instantiate an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).
- Implement a class factory object, which manufactures another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

Note: Delphi wizards automate the creation of COM objects and servers.

CoClasses and Class Factories

A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object provides the services as defined by its interfaces.

CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object's services are requested by a client, a class factory creates an object instance for that particular client. Typically, if another client requests the object's services, the class factory creates another object instance to service the second client. (Clients can also bind to running COM objects that register themselves to support it.)

A CoClass must have a class factory and a class identifier (CLSID) so that it can be instantiated externally, that is, from another module. Using these unique identifiers for CoClasses means that they can be updated whenever new interfaces are implemented in their class. A new interface can modify or add methods without affecting older versions, which is a common problem when using DLLs.

Delphi wizards take care of assigning class identifiers and of implementing and instantiating class factories.

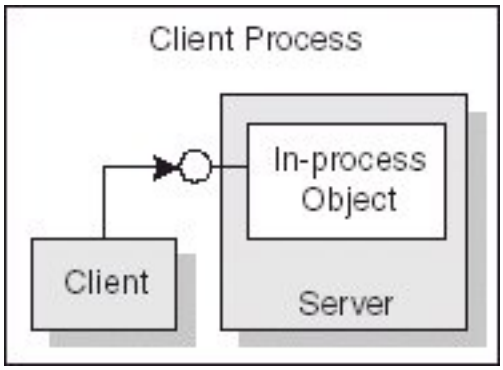
In-process, Out-of-process, and Remote Servers

With COM, a client does not need to know where an object resides, it simply makes a call to an object's interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in

the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:

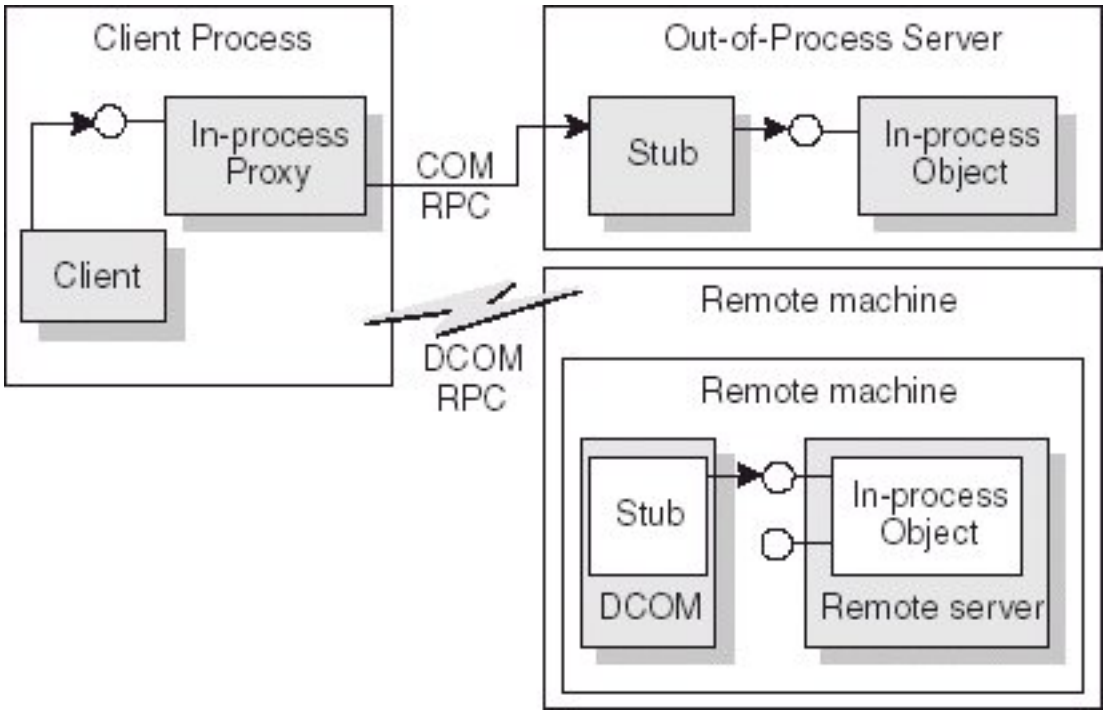
In-process server	A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser. The client communicates with the in-process server using direct calls to the COM interface.
Out-of-process server (or local server)	Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine. The local server uses COM to communicate with the client.
Remote server	A DLL or another application running on a *different machine* from that of the client. For example, a Delphi database application is connected to an application server on another machine in the network. The remote server uses distributed COM (DCOM) to access interfaces and communicate with the application server.

As shown in the following figure, for in-process servers, pointers to the object interfaces are in the same process space as the client, so COM makes direct calls into the object implementation.



Note: This is not always true under COM+. When a client makes a call to an object in a different context, COM+ intercepts the call so that it behaves like a call to an out-of-process server (see below), even if the server is in-process.

As shown in the following figure, when the process is either in a different process or in a different machine altogether, COM uses a proxy to initiate remote procedure calls. The **proxy** resides in the same process as the client, so from the client's perspective, all interface calls look alike. The proxy intercepts the client's call and forwards it to where the real object is running. The mechanism that enables the client to access objects in a different process space, or even different machine, as if they were in their own process, is called marshaling.



The difference between out-of-process and remote servers is the type of interprocess communication used. The proxy uses COM to communicate with an out-of-process server, it uses distributed COM (DCOM) to communicate with a remote machine. DCOM transparently transfers a local object request to the remote object running on a different machine.

Note: For remote procedure calls, DCOM uses the RPC protocol provided by Open Group's Distributed Computing Environment (DCE). For distributed security, DCOM uses the NT LAN Manager (NTLM) security protocol. For directory services, DCOM uses the Domain Name System (DNS).

The Marshaling Mechanism

Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling

- Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed to the proxy. The proxy packs the arguments into a marshaling packet and transmits the structure to the remote object. The object's stub unpacks the packet, pushes the arguments onto the stack, and calls the object's implementation. In essence, the object recreates the client's call in its own address space.

The type of marshaling that occurs depends on what interface the COM object implements. Objects can use a standard marshaling mechanism provided by the *IDispatch* interface. This is a generic marshaling mechanism that enables communication through a system-standard remote procedure call (RPC). For details on the *IDispatch* interface, see Automation Interfaces. Even if the object does not implement *IDispatch*, if it limits itself to automation-compatible types and has a registered type library, COM automatically provides marshaling support.

Applications that do not limit themselves to automation-compatible types or register a type library must provide their own marshaling. Marshaling is provided either through an implementation of the *IMarshal* interface, or by using a separately generated proxy/stub DLL. Delphi does not support the automatic generation of proxy/stub DLLs.

Automation Servers

Sometimes, a server object makes use of another COM object to perform some of its functions. For example, an inventory management object might make use of a separate invoicing object to handle customer invoices. If the inventory management object wants to present the invoice interface to clients, however, there is a problem: Although a client that has the inventory interface can call *QueryInterface* to obtain the invoice interface, when the invoice object was created it did not know about the inventory management object and can't return an inventory interface in response to a call to *QueryInterface*. A client that has the invoice interface can't get back to the inventory interface.

To avoid this problem, some COM objects support **aggregation**. When the inventory management object creates an instance of the invoice object, it passes it a copy of its own *IUnknown* interface. The invoice object can then use that *IUnknown* interface to handle any *QueryInterface* calls that request an interface, such as the inventory interface, that it does not support. When this happens, the two objects together are called an aggregate. The invoice object is called the inner, or contained object of the aggregate, and the inventory object is called the outer object.

Note: In order to act as the outer object of an aggregate, a COM object must create the inner object using the Windows API *CoCreateInstance* or *CoCreateInstanceEx*, passing its *IUnknown* pointer as a parameter that the inner object can use for *QueryInterface* calls.

In order to create an object that can act as the inner object of an aggregate, it must descend from TContainedObject. When the object is created, the *IUnknown* interface of the outer object is passed to the constructor so that it can be used by the *QueryInterface* method on calls that the inner object can't handle.

COM Clients

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the *IDispatch* interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how (or even where) an object provides the services; they simply rely on server objects to provide the services they advertise through their interfaces.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or objects that appear in the container's user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. For details on this process, see Creating COM clients

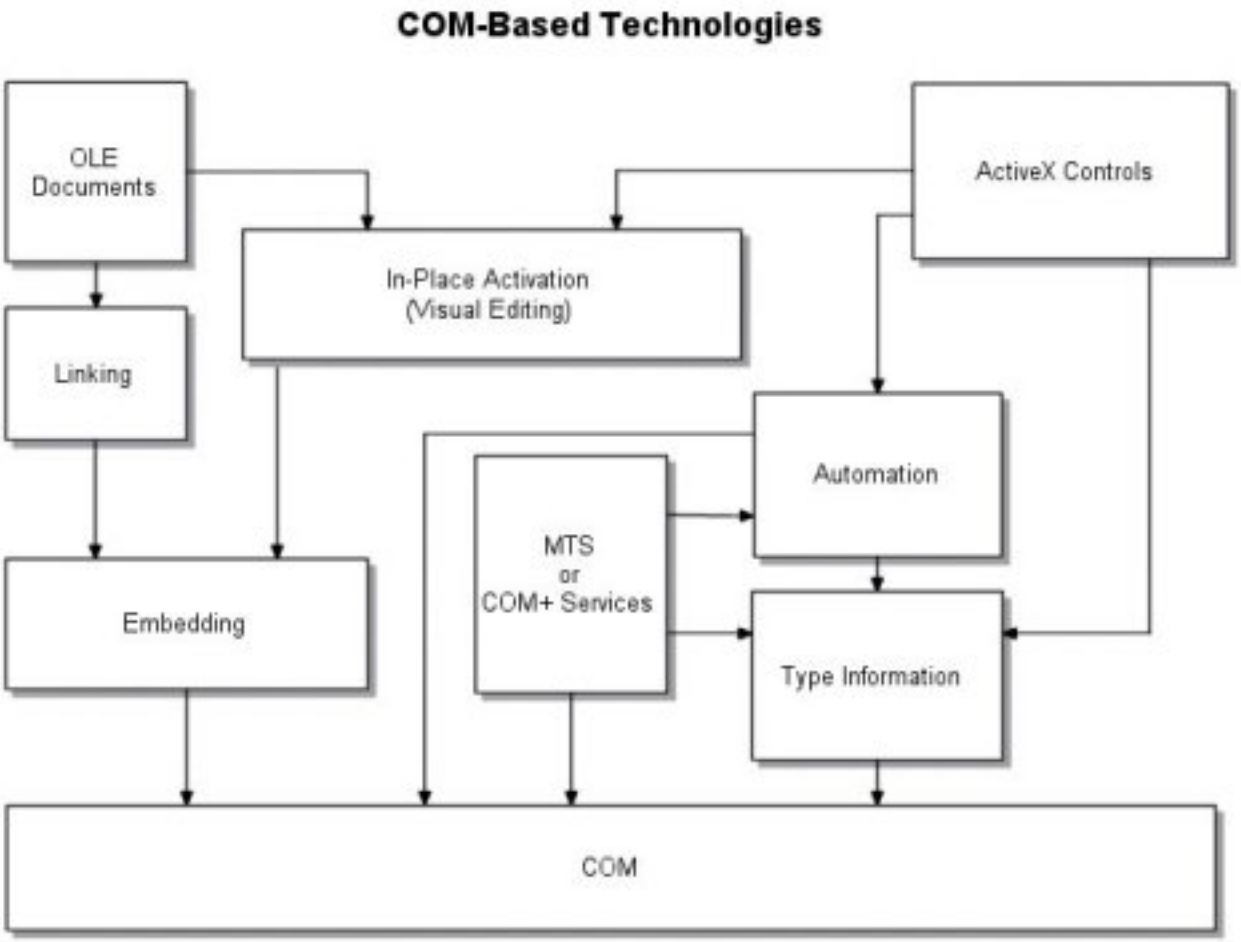
COM Extensions

COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

The following lists some of the services COM extensions currently provide.

Automation servers	Automation refers to the ability of an application to control the objects in another application programmatically. Automation servers are the objects that can be controlled by other executables at runtime.
ActiveX controls	ActiveX controls are specialized in-process servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.
Active Server Pages	Active Server Pages are scripts that generate HTML pages. The scripting language includes constructs for creating and running Automation objects. That is, the Active Server Page acts as an Automation controller.
Active Documents	Objects that support linking and embedding, drag-and-drop, visual editing, and in-place activation. Word documents and Excel spreadsheets are examples of Active Documents.
COM+ Event and event subscription objects	Objects that support the loosely coupled COM+ Events model. Unlike the tightly coupled model used by ActiveX controls, the COM+ Events model allows you to develop event publishers independently of event subscribers.
Type libraries	A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:



COM objects can be visual or non-visual. Some must run in the same process space as their clients; others can run in different processes or remote machines, as long as the objects provide marshaling support. The following table summarizes the types of COM objects that you can create, whether they are visual, process spaces they can run in, how they provide marshaling, and whether they require a type library.

COM object requirements

Object	Visual Object?	Process space	Communication	Type library
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation Server	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the *IDispatch* interface (for out-of process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the *IDispatch* interface	Required
COM+	Occasionally	In-process for MTS,any for COM+	Automatically marshaled via a type library	Required
In-process custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
Other custom interface object	Optionally	In-process,out-of-process, or remote	Automatically marshaled via a type library; otherwise, manually marshaled using custom interfaces	Recommended

Automation Servers

Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and those methods' arguments. Typically, this information is available in a type library. The Automation server can also generate type information dynamically when queried via its *IDispatch* interface (see following).
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the *IDispatch* interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space because the Automation *IDispatch* interface automates the marshaling process. Automation does, however, restrict the types that you can use.

For a list of types that are valid for type libraries in general, and Automation interfaces in particular, see Valid types.

Active Server Pages

The Active Server Page (ASP) technology lets you write simple scripts, called Active Server Pages, that can be launched by clients via a Web server. Unlike ActiveX controls, which run on the client, Active Server Pages run on the server, and return a resulting HTML page to clients.

Active Server Pages are written in Jscript or VB script. The script runs every time the server loads the Web page. That script can then launch an embedded Automation server (or Enterprise Java Bean). For example, you can write an Automation server, such as one to create a bitmap or connect to a database, and this server accesses data that gets updated every time a client loads the Web page.

Active Server Pages rely on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

Delphi wizards let you Create Active Server Pages, which is an Automation object specifically designed to work with an Active Server Page.

ActiveX Controls

ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, TOleContainer, to support linking and embedding of existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note: While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

Type Libraries

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor.

The content of type libraries

Type libraries contain *type information*, which indicates which interfaces exist in which COM objects, and the types and numbers of arguments to the interface methods. These descriptions include the unique identifiers for the CoClasses (CLSIDs) and the interfaces (IIDs), so that they can be properly accessed, as well as the dispatch identifiers (dispIDs) for Automation interface methods and properties.

Type libraries can also contain the following information:

- Descriptions of custom type information associated with custom interfaces
- Routines that are exported by the Automation or ActiveX server, but that are not interface methods
- Information about enumeration, record (structures), unions, alias, and module data types
- References to type descriptions from other type libraries

Creating type libraries

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then running that script through a compiler. However, Delphi automatically generates a type library when you create a COM object (including ActiveX controls, Automation objects, remote data modules, and so on) using any of the wizards on the ActiveX page of the new items dialog. (You can opt not to create a type library when using the COM object wizard.) You can also create a type library by choosing from the main menu, **File** ▸ **New** ▸ **Other**, select the ActiveX folder under Delphi Projects, and in the right pane choose Type Library .

You can view the type library using Delphi's **Type Library Editor**. You can easily edit your type library using the Type Library editor and Delphi automatically updates the corresponding .tlb file (binary type library file) when the type library is saved. For any changes to Interfaces and CoClasses that were created using a wizard, the Type Library editor also updates your implementation files.

When to use type libraries

It is important to create a type library for each set of objects that is exposed to external users, for example,

- ActiveX controls require a type library, which must be included as a resource in the DLL that contains the ActiveX controls.
- Exposed objects that support vtable binding of custom interfaces must be described in a type library because vtable references are bound at compile time. Clients import information about the interfaces from the type library and use that information to compile. For more information about vtable and compile time binding, see Automation interfaces.
- Applications that implement Automation servers should provide a type library so that clients can early bind to it.
- Objects instantiated from classes that support the *IPr ovideClassInfo* interface, such as all descendants of the VCL *TTypedComObject* class, must have a type library.
- Type libraries are not required, but are useful for identifying the objects used with OLE drag-and-drop.

When defining interfaces for internal use only (within an application) you do not need to create a type library. .

Accessing type libraries

The binary type library is normally a part of a resource file (.res) or a stand-alone file with a .tlb file-name extension. When included in a resource file, the type library can be bound into a server (.dll, .ocx, or .exe).

Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Special Type Interfaces

Interface	Description
ITypeLib	Provides methods for accessing a library of type descriptions.
ITypeLib2	Augments *ITypeLib* to include support for documentation strings, custom data, and statistics about the type library.
ITypeInfo	Provides descriptions of individual objects contained in a type library. For example, a browser uses this interface to extract information about objects from the type library.
ITypeInfo2	Augments *ITypeInfo* to access additional type library information, including methods for accessing custom data elements.
ITypeComp	Provides a fast way to access information that compilers need when binding to an interface.

Delphi can import and use type libraries from other applications by choosing Project|Import Type Library. Most of the VCL classes used for COM applications support the essential interfaces that are used to store and retrieve type information from type libraries and from running instances of an object. The VCL class *TTypedComObject* supports interfaces that provide type information, and is used as a foundation for the ActiveX object framework.

Benefits of using type libraries

Even if your application does not require a type library, you can consider the following benefits of using one:

- Type checking can be performed at compile time.
- You can use early binding with Automation, and controllers that do not support vtables or dual interfaces can encode dispIDs at compile time, improving runtime performance.
- Type browsers can scan the library, so clients can see the characteristics of your objects.
- The *RegisterTypeLib* function can be used to register your exposed objects in the registration database.
- The *UnRegisterTypeLib* function can be used to completely uninstall an application's type library from the system registry.

- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.

Using type library tools

The tools for working with type libraries are listed below.

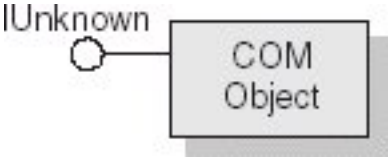
- The TLIBIMP (Type Library Import) tool, which takes existing type libraries and creates Delphi Interface files (_TLB.pas files), is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.
- TRegSvr is a tool for registering and unregistering servers and type libraries, which comes with Delphi. The source to TRegSvr is available as an example in the Demos directory.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- RegSvr32.exe is a standard Windows utility for registering and unregistering servers and type libraries.
- OLEView is a type library browser tool, found on Microsoft's Web site.

Implementing COM Objects with Wizards

Delphi makes it easier to write COM server applications by providing wizards that handle many of the details involved. Delphi provides separate wizards to create the following:

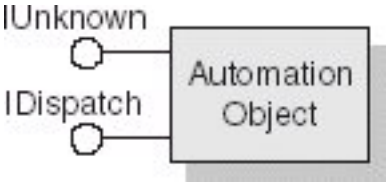
- A simple COM object
- An Automation object
- A COM+ Event Object
- A Type library
- An ActiveX library

The wizards handle many of the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. With a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

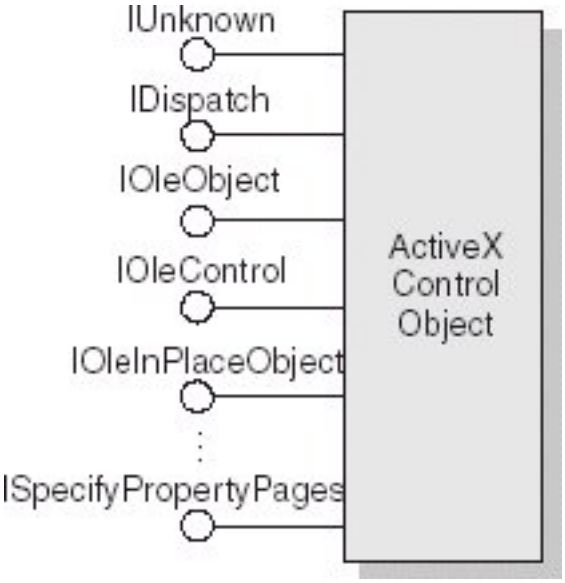


The COM object wizard also provides an implementation for *IDispatch* if you specify that you are creating an object that supports an *IDispatch* descendant.

For Automation and Active Server objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.



For ActiveX control objects and ActiveX forms, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IOleObject*, *IOleControl*, and so on. For a complete list of interfaces, see the reference page for TActiveXControl object.



The following table lists the various wizards and the interfaces they implement:

Delphi wizards for implementing COM, Automation, and ActiveX objects

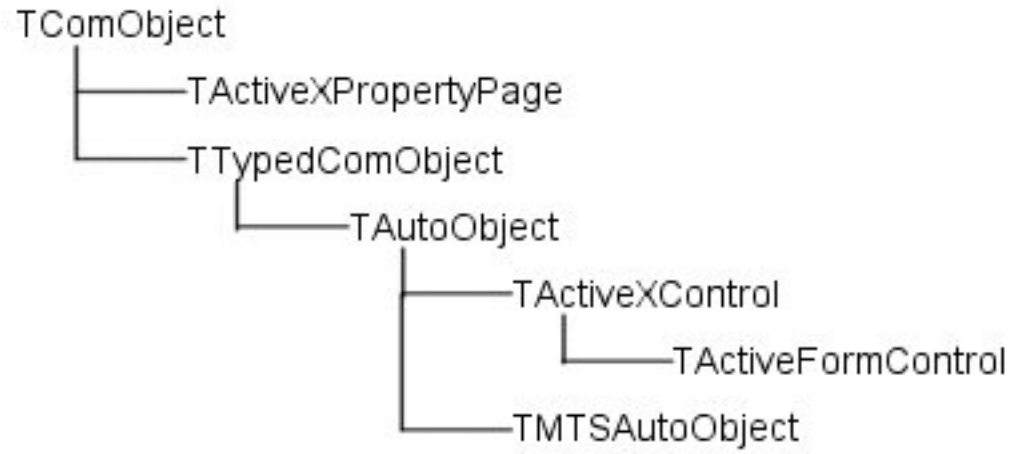
Wizard	Implemented interfaces	What the wizard does
COM server	*IUnknown* (and *IDispatch* if you select a default interface that descends from *IDispatch*)	Exports routines that handle server registration, class registration, loading and unloading the server, and object instantiation. Creates and manages class factories for objects implemented on the server. Provides registry entries for the object that specify the selected threading model. Declares the methods that implement a selected interface, providing skeletal implementations for you to complete. Provides a type library, if requested. Allows you to select an arbitrary interface that is registered in the type library and implement it. If you do this, you must use a type library.
Automation server	*IUnknown*, *IDispatch*	Performs the tasks of a COM server wizard (described above), plus: Implements the interface that you specify, either dual or dispatch. Provides server-side support for generating events, if requested. Provides a type library automatically.
COM+ Event object	None, by default	Creates a COM+ event object that you can define using the Type Library editor. Unlike the other object wizards, the COM+ Event object wizard does not create an implementation unit because event objects

		have no implementation (it is provided by event subscriber objects).
Type Library	None, by default	Creates a new type library and associates it with the active project.
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.

You can add additional COM objects or reimplement an existing implementation. To add a new object, it is easiest to use the wizard a second time. This is because the wizard sets up an association between the type library and an implementation class, so that changes you make in the type library editor are automatically applied to your implementation object.

Code Generated by Wizards

Delphi's wizards generate classes that are derived from the Delphi ActiveX framework (DAX). Despite its name, the Delphi ActiveX framework supports all types of COM objects, not just ActiveX controls. The classes in this framework provide the underlying implementation of the standard COM interfaces for the objects you create using a wizard. The following figure illustrates the objects in the Delphi ActiveX framework:



Each wizard generates an implementation unit that implements your COM server object. The COM server object (the implementation object) descends from one of the classes in DAX:

DAX Base classes for generated implementation classes

Wizard	Base class from DAX	Inherited support
COM server	TTypedComObject	Support for *IUnknown* and *ISupportErrorInfo* interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for reading type library information.
Automation server or Creating Active Server Pages	TAutoObject	Everything provided by *TTypedComObject*, plus: Support for the IDispatch interface. Auto-marshaling support.

Corresponding to the classes in DAX is a hierarchy of class factory objects that handle the creation of these COM objects. The wizard adds code to the initialization section of your implementation unit that instantiates the appropriate class factory for your implementation class.

The wizards also generate a type library and its associated unit, which has a name of the form Project1_TLB. The Project1_TLB unit includes the definitions your application needs to use the type definitions and interfaces defined in the type library. For more information on the contents of this file, see Code generated when you import type library information.

You can modify the interface generated by the wizard using the type library editor. When you do this, the implementation class is automatically updated to reflect those changes. You need only fill in the bodies of the generated methods to complete the implementation.

Working with type libraries

Working with Type Libraries: Overview

Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by a COM object. They provide a way to identify what types of objects and interfaces are available on a server. For a detailed overview on why and when to use type libraries, see Type libraries.

A type library can contain any and all of the following:

- Information about custom data types such as aliases, enumerations, structures, and unions.
- Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass. Each of these descriptions is commonly referred to as type information.
- Descriptions of constants and methods defined in external units.
- References to type descriptions from other type libraries.

By including a type library with your COM application or ActiveX library, you make information about the objects in your application available to other applications and programming tools through COM's type library tools and interfaces.

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then run that script through a compiler. The Type Library editor automates some of this process, easing the burden of creating and modifying your own type libraries.

When you create a COM server of any type (Automation object, remote data module, and so on) using Delphi's wizards, the wizard automatically generates a type library for you (although in the case of the COM object wizard, this is optional). Most of the work you do in customizing the generated object starts with the type library, because that is where you define the properties and methods it exposes to clients: you change the interface of the CoClass generated by the wizard, using the **Type Library Editor**. The Type Library editor automatically updates the implementation unit for your object, so that all you need do is fill in the bodies of the generated methods.

Type Library Editor

The **Type Library Editor** enables developers to examine and create type information for COM objects. Using the **Type Library Editor** can greatly simplify the task of developing COM objects by centralizing the tasks of defining interfaces, CoClasses, and types, obtaining GUIDs for new interfaces, associating interfaces with CoClasses, updating implementation units, and so on.

The **Type Library Editor** outputs two types of file that represent the contents of the type library:

Type Library editor files

File	Description
.TLB file	The binary type library file. By default, you do not need to use this file, because the type library is automatically compiled into the application as a resource. However, you can use this file to explicitly compile the type library into another project or to deploy the type library separately from the .exe or .ocx. For more information, see Opening an existing type library and Deploying type libraries.
_TLB unit	This unit reflects the contents of the type library for use by your application. It contains all the declarations your application needs to use the elements defined in the type library. Although you can open this file in the code editor, you should never edit it—it is maintained by the **Type Library Editor**, so any changes you make will be overwritten by the **Type Library Editor**. For more details on the contents of this file, see Code generated when you import type library information

The following topics describe the **Type Library Editor** in greater detail:

- Parts of the Type Library editor
- Using the Type Library editor

Parts of the Type Library Editor

The main elements of the **Type Library Editor** are described in the following table:

Type Library editor parts

Part	Description
Toolbar	Includes buttons to add new types, CoClasses, interfaces, and interface members to your type library. The toolbar also includes buttons for refreshing your implementation unit, registering the type library, and saving an IDL file with the information in your type library.
Object list pane	Displays all the existing elements in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
Status bar	Displays syntax errors if you try to add invalid types to your type library.
Pages	Display information about the selected object. Which pages appear here depends on the type of object selected.

Toolbar

The **Type Library Editor**'s toolbar located at the top of the **Type Library Editor**, contains buttons that you click to add new objects into your type library.

The first group of buttons let you add elements to the type library. When you click a toolbar button, the icon for that element appears in the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you select, different pages of information appear to the right.

The following table lists the elements you can add to your type library:

Icon	Meaning
	An interface description.
	A dispinterface description.
	A CoClass.

	An enumeration.
	An alias.
	A record.
	A union.
	A module.

When you select one of the elements listed above in the object list pane, the second group of buttons displays members that are valid for that element. For example, when you select Interface, the Method and Property icons in the second box become enabled because you can add methods and properties to your interface definition. When you select Enum, the second group of buttons changes to display the Const member, which is the only valid member for Enum type information.

The following table lists the members that can be added to elements in the object list pane:

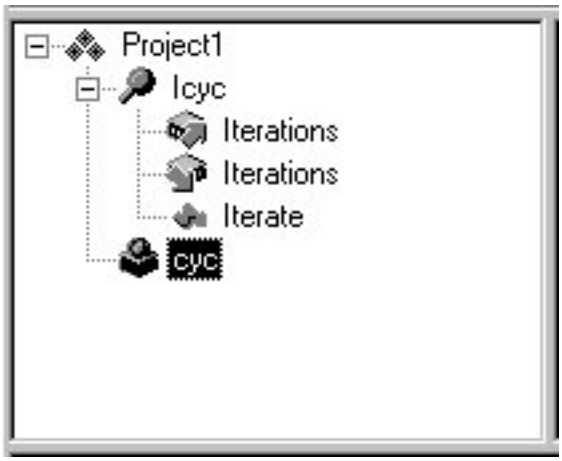
Icon	Meaning
	A method of the interface, dispinterface, or an entry point in a module.
	A property on an interface or dispinterface.
	A write-only property. (available from the drop-down list on the property button)
	A read-write property. (available from the drop-down list on the property button)
	A read-only property. (available from the drop-down list on the property button)
	A field in a record or union.
	A constant in an enum or a module.

The third group of buttons let you refresh, register, or export your type library (save it as an IDL file), as described in Saving and registering type library information.

Object List Pane

The **Object list** pane displays all the elements of the current type library in a tree view. The root of the tree represents the type library itself, and appears as the following icon:

Descending from the type library node are the elements in the type library:



When you select any of these elements (including the type library itself), the pages of type information to the right change to reflect only the relevant information for that element. You can use these pages to edit the definition and properties of the selected element.

You can manipulate the elements in the object list pane by right clicking to get the object list pane context menu. This menu includes commands that let you use the Windows clipboard to move or copy existing elements as well as commands to add new elements or customize the appearance of the **Type Library Editor**.

Status Bar

When editing or saving a type library, syntax, translation errors, and warnings are listed in the **Message** pane.

For example, if you specify a type that the **Type Library Editor** does not support, you will get a syntax error. For a complete list of types supported by the **Type Library Editor**, see Valid types.

Pages of Type Information

When you select an element in the object list pane, pages of type information appear in the **Type Library Editor** that are valid for the selected element. Which pages appear depends on the element selected in the object list panel, as follows:

Type Info element	Page of type information	Contents of page
Type library	Attributes	Name, version, and GUID for the type library, as well as information linking the type library to help.
	Uses	List of other type libraries that contain definitions on which this one depends.
	Flags	Flags that determine how other applications can use the type library.
	Text	All definitions and declarations defining the type library itself (see discussion below).
Interface	Attributes	Name, version, and GUID for the interface, the name of the interface from which it descends, and information linking the interface to help.
	Flags	Flags that indicate whether the interface is hidden, dual, Automation-compatible, and/or extensible.
	Text	The definitions and declarations for the Interface (see discussion below).
Dispinterface	Attributes	Name, version, and GUID for the interface, and information linking it to help.
	Flags	Flags that indicate whether the Dispinterface is hidden, dual, and/or extensible.
	Text	The definitions and declarations for the Dispinterface. (see discussion below).

CoClass	Attributes	Name, version, and GUID for the CoClass, and information linking it to help.
	Implements	A List of interfaces that the CoClass implements, as well as their attributes.
	COM+	The attributes of transactional objects, such as the transaction model, call synchronization, just-in-time activation, object pooling, and so on. Also includes the attributes of COM+ event objects.
	Flags	Flags that indicate various attributes of the CoClass, including how clients can create and use instances, whether it is visible to users in a browser, whether it is an ActiveX control, and whether it can be aggregated (act as part of a composite).
	Text	The definitions and declarations for the CoClass (see discussion below).
Enumeration	Attributes	Name, version, and GUID for the enumeration, and information linking it to help.
	Text	The definitions and declarations for the enumerated type (see discussion below).
Alias	Attributes	Name, version, and GUID for the enumeration, the type the alias represents, and information linking it to help.
	Text	The definitions and declarations for the alias (see discussion below).
Record	Attributes	Name, version, and GUID for the record, and information linking it to help.
	Text	The definitions and declarations for the record (see discussion below).
Union	Attributes	Name, version, and GUID for the union, and information linking it to help.
	Text	The definitions and declarations for the union (see discussion below).
Module	Attributes	Name, version, GUID, and associated DLL for the module, and information linking it to help.
	Text	The definitions and declarations for the module (see discussion below).
Method	Attributes	Name, dispatch ID or DLL entry point, and information linking it to help.
	Parameters	Method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the method, whether this is a default method for the interface, and whether it is replaceable.
	Text	The definitions and declarations for the method (see discussion below).
Property	Attributes	Name, dispatch ID, type of property access method (getter vs. setter), and information linking it to help.
	Parameters	Property access method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the property, whether this is a default for the interface, whether the property is replaceable, bindable, and so on.
	Text	The definitions and declarations for the property access method (see discussion below).
Const	Attributes	Name, value, type (for module consts), and information linking it to help.
	Flags	Flags to indicate how clients can view and use the constant, whether this represents a default value, whether the constant is bindable, and so on.
	Text	The definitions and declarations for the constant (see discussion below).
Field	Attributes	Name, type, and information linking it to help.

	Flags	Flags to indicate how clients can view and use the field, whether this represents a default value, whether the field is bindable, and so on.
	Text	The definitions and declarations for the field (see discussion below).

You can use each of the pages of type information to view or edit the values it displays. Most of the pages organize the information into a set of controls so that you can type in values or select them from a list without requiring that you know the syntax of the corresponding declarations. This can prevent many small mistakes such as typographic errors when specifying values from a limited set. However, you may find it faster to type in the declarations directly. To do this, use the Text page.

All type library elements have a text page that displays the syntax for the element. This syntax appears in an IDL subset of Microsoft Interface Definition Language, or Delphi. See Using Delphi or IDL syntax for details. Any changes you make in other pages of the element are reflected on the text page. If you add code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The **Type Library Editor** generates syntax errors if you add identifiers that are currently not supported by the editor; the editor currently supports only those identifiers that relate to type library support (not RPC support or constructs used by the Microsoft IDL compiler for C++ code generation or marshaling support).

Type Library Elements

The Type Library interface can seem overwhelmingly complicated at first. This is because it represents information about a great number of elements, each of which has its own characteristics. However, many of these characteristics are common to all elements. For example, every element (including the type library itself) has the following:

- A Name, which is used to describe the element and which is used when referring to the element in code.
- A GUID (globally unique identifier), which is a unique 128-bit value that COM uses to identify the element. This should always be supplied for the type library itself and for CoClasses and interfaces. It is optional otherwise.
- A Version number, which distinguishes between multiple versions of the element. This is always optional, but should be provided for CoClasses and interfaces, because some tools can't use them without a version number.
- Information linking the element to a Help topic. These include a Help String, and Help Context or Help String Context value. The Help Context is used for a traditional Windows Help system where the type library has a stand-alone Help file. The Help String Context is used when help is supplied by a separate DLL instead. The Help Context or Help String Context refers to a Help file or DLL that is specified on the type library's **Attributes** page. This is always optional.

Interfaces

An interface describes the methods (and any properties expressed as `get` and `set` functions) for an object that must be accessed through a virtual function table (vtable). If an interface is flagged as dual, it will inherit from *IDispatch*, and your object can provide both early-bound, vtable access, and runtime binding through OLE automation. By default, the type library flags all interfaces you add as dual.

Interfaces can be assigned members: methods and properties. These appear in the object list pane as children of the interface node. Properties for interfaces are represented by the `get` and `set` methods used to read and write the property's underlying data. They are represented in the tree view using special icons that indicate their purpose.

Special Icons for 'get' and 'set' Methods

	A write (set, put) by value property function.
	A read (get) |write (set, put)|write by reference property function.

A read (get) property function.

Note: When a property is specified as Write By Reference, it means it is passed as a pointer rather than by value. Some applications, such a Visual Basic, use Write By Reference, if it is present, to optimize performance. To pass the property only by reference rather than by value, use the property type *By Reference Only*. To pass the property by reference as well as by value, select **Read** ▸ **Write** ▸ **Write By Ref**. To invoke this menu, go to the toolbar and select the arrow next to the property icon.

Once you add the properties or methods using the toolbar button or the object list pane context menu, you describe their syntax and attributes by selecting the property or method and using the pages of type information.

The Attributes page lets you give the property or method a name and dispatch ID (so that it can be called using IDispatch). For properties, you also assign a type. The function signature is created using the Parameters page, where you can add, remove, and rearrange parameters, set their type and any modifiers, and specify function return types.

Note: Members of interfaces that need to raise exceptions should return an HRESULT and specify a return value parameter (PARAM_RETVAL) for the actual return value. Declare these methods using the **safecall** calling convention.

Note that when you assign properties and methods to an interface, they are implicitly assigned to its associated CoClass. This is why the Type Library editor does not let you add properties and methods directly to a CoClass.

Dispinterfaces

Interfaces are more commonly used than dispinterfaces to describe the properties and methods of an object. Dispinterfaces are only accessible through dynamic binding, while interfaces can have static binding through a vtable.

You can add methods and properties to dispinterfaces in the same way you add them to interfaces. However, when you create a property for a dispinterface, you can't specify a function kind or parameter types.

CoClasses

A CoClass describes a unique COM object that implements one or more interfaces. When defining a CoClass, you must specify which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events. Note that you do not add properties or methods to a CoClass in the Type Library editor. Properties and methods are exposed to clients by interfaces, which are associated with the CoClass using the Implements page.

Type definitions

Enumerations, aliases, records, and unions all declare types that can then be used elsewhere in the type library.

Enums consist of a list of constants, each of which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default. You can add constants to your enumeration by selecting the enumeration in the object list pane and clicking the Const button on the toolbar or selecting **New** ▸ **Const** command from the object list pane context menu.

Note: It is strongly recommended that you provide help strings for your enumerations to make their meaning clearer. The following is a sample entry of an enumeration type for a mouse button and includes a help string for each enumeration element.

```
[Delphi]
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
```

```
[C++]
typedef enum TxMouseButton
{
[helpstring("mbLeft")]
mbLeft = 0,
[helpstring("mbRight)]
mbRight = 1.
[helpstring("mbMiddle)]
mbMiddle = 2
} TxMouseButton;
```

An alias creates an alias (type definition) for a type. You can use the alias to define types that you want to use in other type info such as records or unions. Associate the alias with the underlying type definition by setting the Type attribute on the Attributes page.

A record consists of a list of structure members or fields. A union is a record with only a variant part. Like a record, a union consists of a list of structure members or fields. However, unlike the members of records, each member of a union occupies the same physical address, so that only one logical value can be stored.

Add the fields to a record or union by selecting it in the object list pane and clicking the field button in the toolbar or right clicking and choosing field from the object list pane context menu. Each field has a name and a type, which you assign by selecting the field and assigning values using the Attributes page. Records and unions can be defined with an optional tag.

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Modules

A module defines a group of functions, typically a set of DLL entry points. You define a module by

- Specifying a DLL that it represents on the attributes page.
- Adding methods and constants using the toolbar or the object list pane context menu. For each method or constant, you must then define its attributes by selecting the it in the object list pane and setting the values on the Attributes page.

For module methods, you must assign a name and DLL entry point using the attributes page. Declare the function's parameters and return type using the parameters page.

For module constants, use the Attributes page to specify a name, type, and value.

Note: The **Type Library Editor** does not generate any declarations or implementation related to a module. The specified DLL must be created as a separate project.

Using the Type Library Editor

Using the type library editor, you can create new type libraries or edit existing ones. Typically, an application developer uses a wizard to create the objects that are exposed in the type library, letting Delphi generate the type library automatically. Then, the automatically-generated type library is opened in the Type Library editor so that the interfaces can be defined (or modified), type definitions added, and so on.

However, even if you are not using a wizard to define the objects, you can use the Type Library editor to define a new type library. In this case, you must create any implementation classes yourself, because the Type Library editor does not generate code for CoClasses that were not associated with a type library by a wizard.

The editor supports a subset of valid types in a type library

The following topics describe how to:

- Create a new type library
- Open an existing type library
- Add an interface to the type library
- Modify an interface
- Add properties and methods to the type library
- Add a CoClass to the type library
- Add an interface to a CoClass
- Add an enumeration to the type library
- Add an alias to the type library
- Add a record or union to the type library
- Add a module to the type library
- Save and register type library information

Valid Types

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Delphi. Specify the language you want to use in the Environment options dialog.

The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or Dispinterface flag checked. These are the types that COM can marshal via the type library automatically.

Delphi type	IDL type	variant type	Automation compatible	Description
Smallint	short	VT_I2	Yes	2-byte signed integer
Integer	long	VT_I4	Yes	4-byte signed integer
Single	single	VT_R4	Yes	4-byte real
Double	double	VT_R8	Yes	8-byte real
Currency	CURRENCY	VT_CY	Yes	currency
TDateTime	DATE	VT_DATE	Yes	date
WideString	BSTR	VT_BSTR	Yes	binary string
IDispatch	IDispatch	VT_DISPATCH	Yes	pointer to IDispatch interface
SCODE	SCODE	VT_ERROR	Yes	Ole Error Code
WordBool	VARIANT_BOOL	VT_BOOL	Yes	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Yes	Ole Variant
IUnknown	IUnknown	VT_UNKNOWN	Yes	pointer to IUnknown interface
Shortint	byte	VT_I1	No	1 byte signed integer
Byte	unsigned char	VT_UI1	Yes	1 byte unsigned integer

Word	unsigned short	VT_UI2	Yes*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	Yes*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed integer
Largeuint	uint64	VT_UI8	No	8 byte unsigned integer
SYSINT	int	VT_INT	Yes*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Yes*	system dependent unsigned integer
HResult	HRESULT	VT_HRESULT	No	32 bit error code
Pointer		VT_PTR -> VT_VOID	No	untyped pointer
SafeArray	SAFEARRAY	VT_SAFEARRAY	No	OLE Safe Array
PChar	LPSTR	VT_LPSTR	No	pointer to Char
PWideChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar

* Word, LongWord, SYSINT, and SYSUINT may be Automation-compatible with some applications.

See safe arrays for more information about the SAFEARRAY Variant type.

Note: The Byte (VT_UI1) is Automation-compatible, but is not allowed in a Variant or OleVariant since many Automation servers do not handle this value correctly.

Besides these IDL types, any interfaces and types defined in the library or defined in referenced libraries can be used in a type library definition.

The Type Library editor stores type information in the generated type library (.TLB) file in binary form.

If a parameter type is specified as a Pointer type, the Type Library editor usually translates that type into a variable parameter. When the type library is saved, the variable parameter's associated ElemDesc's IDL flags are marked IDL_FIN or IDL_FOUT.

Often, ElemDesc IDL flags are not marked by IDL_FIN or IDL_FOUT when the type is preceded with a Pointer. Or, in the case of dispinterfaces, IDL flags are not typically used. In these cases, you may see a comment next to the variable identifier such as {IDL_None} or {IDL_In}. These comments are used when saving a type library to correctly mark the IDL flags.

SafeArrays

COM requires that arrays be passed via a special data type known as a SafeArray. You can create and destroy SafeArrays by calling special COM functions to do so, and all elements within a SafeArray must be valid automation-compatible types. The Delphi compiler has built-in knowledge of COM SafeArrays and automatically calls the COM API to create, copy, and destroy SafeArrays.

In the **Type Library Editor**, a *SafeArray* must specify the type of its elements. For example, the following line from the text page declares a method with a parameter that is a *SafeArray* with an element type of Integer:

```
[Delphi]
procedure HighLightLines(Lines: SafeArray of Integer);
```

```
[C++]
HRESULT _stdcall HighlightLines(SAFEARRAY(long) Lines);
```

Note: Although you must specify the element type when declaring a *SafeArray* type in the **Type Library Editor**, the declaration in the generated _TLB unit does not indicate the element type.

Using Object Pascal or IDL Syntax

The Text page of the Type Library editor displays your type information in one of two ways:

- Using an extension of Delphi syntax.
- Using the Microsoft IDL.

You can select which language you want to use by changing the setting in the Environment Options dialog. Choose **Tools** ▸ **Options**, and specify either Pascal or IDL as the Language on the Type Library page of the dialog.

Note: The choice of Delphi or IDL syntax also affects the choices available on the parameters attributes page.

Like Delphi applications in general, identifiers in type libraries are case insensitive. They can be up to 255 characters long, and must begin with a letter or an underscore (_).

Attribute specifications

Delphi has been extended to allow type libraries to include attribute specifications. Attribute specifications appear enclosed in square brackets and separated by commas. Each attribute specification consists of an attribute name followed (if appropriate) by a value.

The following table lists the attribute names and their corresponding values.

Attribute syntax

Attribute name	Example	Applies to
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	members except CoClass members
control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass members
defaultbind	[defaultbind]	members except CoClass members
defaultcollection	[defaultcollection]	members except CoClass members
defaultvtbl	[defaultvtbl]	CoClass members
dispid	[dispid]	members except CoClass members
displaybind	[displaybind]	members except CoClass members
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	type library
helpcontext	[helpcontext 2005]	anything except CoClass members and parameters
helpstring	[helpstring 'payroll interface']	anything except CoClass members and parameters
helpstringcontext	[helpstringcontext $17]	anything except CoClass members and parameters
hidden	[hidden]	anything except parameters

immediatebind	[immediatebind]	members except CoClass members
lcid	[lcid $324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	members except CoClass members
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	members except CoClass members
propput	[propput]	members except CoClass members
propputref	[propputref]	members except CoClass members
public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass members
replaceable	[replaceable]	anything except CoClass members and parameters
requestedit	[requestedit]	members except CoClass members
restricted	[restricted]	anything except parameters
source	[source]	all members
uidefault	[uidefault]	members except CoClass members
usesgetlasterror	[usesgetlasterror]	members except CoClass members
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	type library, typeinfo (required)
vararg	[vararg]	members except CoClass members
version	[version 1.1]	type library, typeinfo

Interface syntax

The Delphi syntax for declaring interface type information has the form

```
interfacename = interface[(baseinterface)] [attributes]
functionlist
[propertymethodlist]
end;
```

For example, the following text declares an interface with two methods and one property:

```
Interface1 = interface (IDispatch)
  [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
  function Calculate(optional seed:Integer=0): Integer;
  procedure Reset;
  procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
  function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',version 1.0]
interface Interface1 :IDispatch
{
  long Calculate([in, optional, defaultvalue(0)] long seed);
  void Reset(void);
  [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
  [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Dispatch interface syntax

The Delphi syntax for declaring dispinterface type information has the form

```
dispinterfacename = dispinterface [attributes]
functionlist
[propertylist]
end;
```

For example, the following text declares a dispinterface with the same methods and property as the previous interface:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring 'dispatch interface for MyObj'
  function Calculate(seed:Integer): Integer [dispid 1];
  procedure Reset [dispid 2];
  property Range: Integer [dispid 3];
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring "dispatch interface for MyObj"
dispinterface Interface1
{
  methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
  properties:
    [id(3)] int Value;
};
```

CoClass syntax

The Delphi syntax for declaring CoClass type information has the form

```
classname = coclass(interfacename[interfaceattributes], ...); [attributes];
```

For example, the following text declares a coclass for the interface *IMyInt* and dispinterface *DmyInt*:

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring "A class",
  appobject]
coclass myapp
{
  methods:
  [source] interface IMyInt);
  dispinterface DMyInt;
};
```

Enum syntax

The Delphi syntax for declaring Enum type information has the form

```
enumname = ([attributes] enumlist);
```

For example, the following text declares an enumerated type with three values:

```
location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
         helpstring 'location of booth']
  Inside = 1 [helpstring 'Inside the pavillion'];
  Outside = 2 [helpstring 'Outside the pavillion'];
  Offsite = 3 [helpstring 'Not near the pavillion'];);
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "location of booth"]
typedef enum
{
  [helpstring "Inside the pavillion"] Inside = 1,
  [helpstring "Outside the pavillion"] Outside = 2,
  [helpstring "Not near the pavillion"] Offsite = 3
} location;
```

Alias syntax

The Delphi syntax for declaring Alias type information has the form

```
aliasname = basetype[attributes];
```

For example, the following text declares DWORD as an alias for integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record syntax

The Delphi syntax for declaring Record type information has the form

```
recordname = record [attributes] fieldlist end;
```

For example, the following text declares a record:

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                helpstring 'Task description']
  ID: Integer;
  StartDate: TDate;
  EndDate: TDate;
  Ownername: WideString;
  Subtasks: safearray of Integer;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "Task description"]
typedef struct
{
  long ID;
  DATE StartDate;
  DATE EndDate;
  BSTR Ownername;
  SAFEARRAY (int) Subtasks;
} Tasks;
```

Union syntax

The Delphi syntax for declaring Union type information has the form

```
unionname = record [attributes]
case Integer of
  0: field1;
  1: field2;
  ...
end;
```

For example, the following text declares a union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                  helpstring "item description"]
case Integer of
```

```
  0: (Name: WideString);
  1: (ID: Integer);
  3: (Value: Double);
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "item description"]
typedef union
{
  BSTR Name;
  long ID;
  double Value;
  } MyUnion;
```

Module syntax

The Delphi syntax for declaring Module type information has the form

```
modulename = module constants entrypoints end;
```

For example, the following text declares the type information for a module:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                   dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius: Double): Double [ entry 1 ]; stdcall;
  function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        dllname("circle.dll")]
module MyModule
{
  double PI = 3.14159;
  [entry(1)] double _stdcall area([in] double radius);
  [entry(2)] double _stdcall circumference([in] double radius);
};
```

Creating a New Type Library

You may want to create a type library that is independent of a particular COM object. For example, you might want to define a type library that contains type definitions that you use in several other type libraries. You can then create a type library of basic definitions and add it to the uses page of other type libraries.

You can also create a type library for an object that is not yet implemented. Once the type library contains the interface definition, you can use the COM object wizard to generate a CoClass and implementation.

To create a new type library

1. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
2. Choose the **ActiveX** folder under **Delphi Projects**
3. Select the **Type Library** icon in the right pane.
4. Choose OK.

   Enter a name for the type library.
5. Continue by adding elements to your type library.

Opening an Existing Type Library

When you use the wizards to create an Automation object, COM object, transactional object, or a remote data module, a type library is automatically created with an implementation unit. In addition, you may have type libraries that are associated with other products (servers) that are available on your system.

To open a type library that is not currently part of your project,

1. Choose **File** ▸ **Open** from the main menu in the IDE.
2. In the Open dialog box, set the File Type to type library.
3. Navigate to the desired type library files and choose Open.

To open a type library associated with the current project,

Choose **View** ▸ **Type Library**.

Now, you can add interfaces, CoClasses, and other elements of the type library such as enumerations, properties, and methods.

Note: Changes you make to any type library information with the **Type Library Editor** can be automatically reflected in the associated implementation class. If you want to review the changes before they are added, be sure that the Apply Updates dialog is on. It is on by default and can be changed in the setting, "Display updates before refreshing," on the**Tools** ▸ **Options** ▸ **Delphi Options** ▸ **Type Library** page.

Tip: When writing client applications, you do not need to open the type library. You only need the *Project*_TLB unit that the **Type Library Editor** creates from a type library, not the type library itself. You can add this file directly to a client project, or, if the type library is registered on your system, you can use the Import Type Library dialog (**Component** ▸ **Import Type Library**).

Adding an Interface to the Type Library

To add an interface

1. On the toolbar, click on the interface icon.

   An interface is added to the object list pane prompting you to add a name.
2. Type a name for the interface.

The new interface contains default attributes that you can modify as needed.

You can add properties (represented by getter/setter functions) and methods to suit the purpose of the interface.

Modifying an Interface Using the Type Library

There are several ways to modify an interface or dispinterface once it is created.

- You can change the interface's attributes using the page of type information that contains the information you want to change. Select the interface in the **Object List** pane and then use the controls on the appropriate page of type information. For example, you may want to change the parent interface using the attributes page, or use the flags page to change whether or not it is a dual interface.
- You can edit the interface declaration directly by selecting the interface in the object list pane and then editing the declarations on the **Text** page.
- You can Add properties and methods to the interface.
- You can modify the properties and methods already in your interface by changing their type information.
- You can associate it with a CoClass by selecting the CoClass in the object list pane, right-clicking on the Implements page, and choosing Insert Interface.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the **Type Library Editor** to apply your changes to the implementation file by clicking the **Refresh** button on the toolbar. If you have the Apply Updates dialog enabled, the **Type Library Editor** notifies you before updating the sources and warns you of potential problems. For example, if you rename an event interface by mistake, you may get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You also get a TODO comment in your source file immediately above it.

Warning: If you ignore this warning and TODO comment, the code will not compile.

Adding Properties and Methods to the Type Library

To add properties or methods to an interface or dispinterface

1. Select the interface, and choose either a **property** or **method** icon from the toolbar. If you are adding a property, you can click directly on the property icon to create a read/write property (with both a getter and a setter), or click the down arrow to display a menu of property types.

   The property access method members or method member is added to the object list pane, prompting you to add a name.

2. Type a name for the member.

The new member contains default settings on its attributes, parameters, and flags pages that you can modify to suit the member. For example, you will probably want to assign a type to a property on the attributes page. If you are adding a method, you will probably want to specify its parameters on the parameters page.

As an alternate approach, you can add properties and methods by typing directly into the text page using Delphi or IDL syntax. For example, if you are working in Delphi syntax, you can type the following property declarations into the text page of an interface:

```
Interface1 = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
 function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Add this
 function AutoSize: WordBool [propget, dispid $00000001]; safecall; // And this
 procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // And this
end;
```

If you are working in IDL, you can add the same declarations as follows:

```
[
   uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
   version(1.0),
   dual,
   oleautomation
]
interface Interface1: IDispatch
{ // Add everything between the curly braces
[propget, id(0x00000002)]
   HRESULT _stdcall AutoSelect([out, retval] long Value );
   [propget, id(0x00000003)]
   HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
   [propput, id(0x00000003)]
   HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

After you have added members to an interface using the interface text page, the members appear as separate items in the object list pane, each with its own attributes, flags, and parameters pages. You can modify each new property or method by selecting it in the object list pane and using these pages, or by making edits directly in the text page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the **Type Library Editor** to apply your changes to the implementation file by clicking the **Refresh** button on the toolbar. The **Type Library Editor** adds new methods to your implementation class to reflect the new members. You can then locate the new methods in implementation unit's source code and fill in their bodies to complete the implementation.

If you have the Apply Updates dialog enabled, the **Type Library Editor** notifies you of all changes before updating the sources and warns you of potential problems.

Adding a CoClass to the Type Library

The easiest way to add a CoClass to your project is to choose **File** ▸ **New** ▸ **Other** from the main menu in the IDE and use the appropriate wizard on the ActiveX page of the **New Items** dialog. The advantage to this approach is that, in addition to adding the CoClass and its interface to the type library, the wizard adds an implementation unit and updates the project file to include the new implementation unit in its uses clause.

If you are not using a wizard, however, you can create a CoClass by clicking the CoClass icon on the toolbar and then specifying its attributes. You will probably want to give the new CoClass a name (on the Attributes page), and may want to use the Flags page to indicate information such as whether the CoClass is an application object, whether it represents an ActiveX control, and so on.

Note: When you add a CoClass to a type library using the toolbar instead of a wizard, you must generate the implementation for the CoClass yourself and update it by hand every time you change an element on one of the CoClass' interfaces.

You can't add members directly to a CoClass. Instead, you implicitly add members when you add an interface to the CoClass.

Adding an Interface to a CoClass

CoClasses are defined by the interfaces they present to clients. While you can add any number of properties and methods to the implementation class of a CoClass, clients can only see those properties and methods that are exposed by interfaces associated with the CoClass.

To associate an interface with a CoClass, right-click in the **Implements** page for the class and choose Insert Interface to display a list of interfaces from which you can choose. The list includes interfaces that are defined in the current type library and those defined in any type libraries that the current type library references. Choose an interface you want the class to implement. The interface is added to the page with its GUID and other attributes.

If the CoClass was generated by a wizard, the **Type Library Editor** automatically updates the implementation class to include skeletal methods for the methods (including property access methods) of any interfaces you add this way. If you have the Apply Updates dialog enabled, the **Type Library Editor** notifies you before updating the sources and warns you of potential problems.

Adding an Enumeration to the Type Library

To add enumerations to a type library

1. On the toolbar, click on the enum icon.

   An enum type is added to the **Object List** pane .

2. Type a name for the enumeration.

   The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by right clicking the enum and selecting the **New** ▸ **Const** button . Then, select each enumerated value and assign it a name (and possibly a value) using the attributes page.

Once you have added an enumeration, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the enumeration as the type for a property or parameter.

Adding an Alias to the Type Library

To add an alias to a type library

1. On the toolbar, click on the alias icon.

   An alias type is added to the object list pane .

2. Type a name for the alias.

   By default, the new alias stands for a Long Integer type. Use the **Attributes** page to change this to the type you want the alias to represent.

Once you have added an alias, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the alias as the type for a property or parameter.

Adding a Record or Union to the Type Library

To add a record or union to a type library

1. On the toolbar, click on the record icon or the union icon.

   The selected type element is added to the object list pane .
2. Type a name for the record or union.

   At this point, the new record or union contains no fields.
3. With the record or union selected in the object list pane, click on the field icon in the toolbar. Specify the field's name and type, using the **Attributes** page.
4. Repeat step 3 for as many fields as you need.

Once you have defined the record or union, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the record or union as the type for a property or parameter.

Adding a Module to the Type Library

To add a module to a type library

1. On the toolbar, click on the module icon.

   The selected module is added to the object list pane .
2. Type a name for the module.
3. On the **Attributes** page, specify the name of the DLL whose entry points the Module represents.
4. Add any methods from the DLL you specified in step 3 by clicking on the **Method** icon in the toolbar and then using the attributes pages to describe the method.
5. Add any constants you want the module to define by clicking on the **Const** icon on the toolbar. For each constant, specify a name, type, and value.

Saving and Registering Type Library Information

After modifying your type library, you'll want to save and register the type library information.

Saving the type library automatically updates:

- The binary type library file (.tlb extension).
- The *Project*_TLB unit that represents its contents
- The implementation code for any CoClasses that were generated by a wizard.

Note: The type library is stored as a separate binary (.TLB) file, but is also linked into the server (.EXE, DLL, or .OCX).

The **Type Library Editor** gives you options for storing your type library information. Which way you choose depends on what stage you are at in implementing the type library:

- Save, to save both the .TLB and the *Project*_TLB unit to disk. (It is accessible through **File** ▸ **Save** in the IDE.)
- Refresh, to update the type library units in memory only.

- Register, to add an entry for the type library in your system's Windows registry. This is done automatically when the server with which the .TLB is associated is itself registered.
- Export, to save a .IDL file that contains the type and interface definitions in IDL syntax.

All the above methods perform syntax checking. When you refresh, register, or save the type library, Delphi automatically updates the implementation unit of any CoClasses that were created using a wizard. Optionally, you can review these updates before they are committed, if you have the **Type Library Editor** option, Apply Updates on.

Apply Updates Dialog

The **Apply Updates** dialog appears when you refresh, register, or save the type library if you have selected **Display updates before refreshing** in the **Tools** ▸ **Options** ▸ **Type Library** page (which is not checked off by default).

Without this option, the **Type Library Editor** automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library.

The **Apply Updates** dialog will warn you about potential errors, and will insert TODO comments in your source file. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

Note: If you ignore this warning and TODO comment, the code will not compile.

Saving a Type Library

Saving a type library:

- Performs a syntax and validity check.
- Saves information out to a .TLB file.
- Saves information out to the *Project*_TLB unit.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To save the type library, choose **File** ▸ **Save** from the Delphi main menu.

Refreshing the Type Library

Refreshing the type library

- Performs a syntax check.
- Regenerates the Delphi type library units in memory only. It does not save any files to disk.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To refresh the type library choose the **Refresh** icon on the **Type Library Editor toolbar**.

Note: If you have renamed items in the type library, refreshing the implementation may create duplicate entries. In this case, you must move your code to the correct entry and delete any duplicates. Similarly, if you delete items in the type library, refreshing the implementation does not remove them from CoClasses (under the assumption that you are merely removing them from visibility to clients). You must delete these items manually in the implementation unit if they are no longer needed.

Registering the Type Library

Typically, you do not need to explicitly register a type library because it is registered automatically when you register your COM server application (see Registering a COM object). However, when you create a type library using the **Type Library wizard**, it is not associated with a server object. In this case, you can register the type library directly using the toolbar.

Registering the type library,

- Performs a syntax check
- Adds an entry to the Windows Registry for the type library

To register the type library, choose the **Register** icon on the **Type Library Editor toolbar**.

Exporting an IDL File

Exporting the type library,

- Performs a syntax check.
- Creates a Microsoft IDL file that contains the type information declarations.

To export the type library, choose the **Export** icon on the **Type Library Editor** toolbar.

Deploying Type Libraries

By default, when you have a type library that was created as part of an Automation server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource.

You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer.

Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in the executable (.DLL, .OCX, or .EXE) file.

When you make type libraries other than the primary project type library available to application developers, the type libraries can be in any of the following forms:

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.
- A resource. This resource should have the type TYPELIB and an integer ID. If you choose to build type libraries with a resource compiler, it must be declared in the resource (.RC) file as follows:

```
1 typelib mylib1.tlb
2 typelib mylib2.tlb
```

Creating COM clients

Creating COM Clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are applications that control an Automation server (Automation controllers) and applications that host an ActiveX control (ActiveX containers).

At first glance these two types of COM client are very different: The typical Automation controller launches an external server EXE and issues commands to make that server perform tasks on its behalf. The Automation server is usually nonvisual and out-of-process. The typical ActiveX client, on the other hand, hosts a visual control, using it much the same way you use any control on the Component palette. ActiveX servers are always in-process servers.

However, the task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server object and uses its properties and methods. Developer Studio 2006 makes this particularly easy by letting you wrap the server CoClass in a component on the client, which you can even install on the Component palette.

When writing a COM client, you must understand the interface that the server exposes to clients, just as you must understand the properties and methods of a component from the Component palette to use it in your application. This interface (or set of interfaces) is determined by the server application, and typically published in a type library. For specific information on a particular server application's published interfaces, you should consult that application's documentation.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information.

Once you have imported the type information, you can write code to control the imported object.

Note: You can also query the type information directly using COM APIs, but Developer Studio 2006 provides no special support for this.

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. These are discussed in Creating Clients for Servers That Do Not Have a Type Library.

Importing Type Library Information

To make information about the COM server available to your client application, you must import the information about the server that is stored in the server's type library. Your application can then use the resulting generated classes to control the server object.

There are two ways to import type library information:

- You can use the **Import Component dialog** to import all available information about the server types, objects, and interfaces. This is the most general method, because it lets you import information from any type library and can optionally generate component wrappers for all creatable CoClasses in the type library that are not flagged as Hidden, Restricted, or PreDeclID.
- You can also use the **Import Component dialog** if you are importing from the type library of an ActiveX control. This imports the same type information, but only creates component wrappers for CoClasses that represent ActiveX controls.
- You can use the command line utility tlibimp.exe which provides additional configuration options not available from within the IDE.
- A type library generated using a wizard is automatically imported using the same mechanism as the import type library menu item.

Regardless of which method you choose to import type library information, the resulting dialog creates a unit with the name *TypeLibName*_TLB, where *TypeLibName* is the name of the type library. This file contains declarations for the classes, types, and interfaces defined in the type library. By including it in your project, those definitions are available to your application so that you can create objects and call their interfaces. This file may be recreated by the IDE from time to time; as a result, making manual changes to the file is not recommended.

In addition to adding type definitions to the *TypeLibName*_TLB unit, the dialog can also create VCL class wrappers for any CoClasses defined in the type library. When you use the Import Type Library dialog, these wrappers are optional. When you use the Import ActiveX dialog, they are always generated for all CoClasses that represent controls.

The generated class wrappers represent the CoClasses to your application, and expose the properties and methods of its interfaces. If a CoClass supports the interfaces for generating events (*IConnectionPointContainer* and *IConnectionPoint*), the VCL class wrapper creates an event sink so that you can assign event handlers for the events as simply as you can for any other component. If you tell the dialog to install the generated VCL classes on the **Tool Palette**, you can use the **Object Inspector** to assign property values and event handlers.

Note: The **Import Component dialog** does not create class wrappers for COM+ event objects. To write a client that responds to events generated by a COM+ event object, you must create the event sink programmatically. This process is described in Handling COM+ events.

For more details about the code generated when you import a type library, see Code generated when you import type library information.

Code Generated When You Import Type Library Information

Once you import a type library, you can view the generated *TypeLibName*_TLB unit. At the top, you will find the following:

First, constant declarations giving symbolic names to the GUIDS of the type library and its interfaces and CoClasses. The names for these constants are generated as follows:

- the GUID for the type library has the form LBID_*TypeLibName*, where *TypeLibName* is the name of the type library.
- The GUID for an interface has the form IID_*InterfaceName*, where *InterfaceName* is the name of the interface.
- The GUID for a dispinterface has the form DIID_*InterfaceName*, where *InterfaceName* is the name of the dispinterface.
- The GUID for a CoClass has the form CLASS_*ClassName*, where *ClassName* is the name of the CoClass.
- The compiler directive VARPROPSETTER will be on. This allows the use of the keyword var in the parameter list of property setter methods. This disables a compiler optimization that would cause parameters to be passed

by value instead of by reference. The VARPROPSETTER directive must be on, when creating TLB units for components written in a language other than Delphi.

Second, declarations for the CoClasses in the type library. These map each CoClass to its default interface.

Third, declarations for the interfaces and dispinterfaces in the type library.

Fourth, declarations for a creator class for each CoClass whose default interface supports VTable binding. The creator class has two class methods, *Create* and *CreateRemote*, that can be used to instantiate the CoClass locally (*Create*) or remotely (*CreateRemote*).These methods return the default interface for the CoClass.

These declarations provide you with what you need to create instances of the CoClass and access its interface. All you need do is add the generated *TypeLibName*_TLB.pas file to the uses clause of the unit where you wish to bind to a CoClass and call its interfaces.

Note: This portion of the *TypeLibName*_TLB unit is also generated when you use the Type Library editor or the command-line utility TLIBIMP.

If you want to use an ActiveX control, you also need the generated VCL wrapper in addition to the declarations described above. The VCL wrapper handles window management issues for the control. You may also have generated a VCL wrapper for other CoClasses in the Import Type Library dialog. These VCL wrappers simplify the task of creating server objects and calling their methods. They are especially recommended if you want your client application to respond to events.

The declarations for generated VCL wrappers appear at the bottom of the interface section. Component wrappers for ActiveX controls are descendants of TOleControl. Component wrappers for Automation objects descend from TOleServer. The generated component wrapper adds the properties, events, and methods exposed by the CoClass's interface. You can use this component like any other VCL component.

Warning: You should not edit the generated *TypeLibName*_TLB unit. It is regenerated each time the type library is refreshed, so any changes will be overwritten.

Note: For the most up-to-date information about the generated code, refer to the comments in the automatically-generated *TypeLibName*_TLB unit.

Controlling an Imported Object

After importing type library information, you are ready to start programming with the imported objects. How you proceed depends in part on the objects, and in part on whether you have chosen to create component wrappers. There are two basic approaches:

- Using component wrappers.
- Writing client code based on type library definitions.

Using Component Wrappers

If you generated a component wrapper for your server object, writing your COM client application is not very different from writing any other application that contains VCL components. The server object's properties, methods, and events are already encapsulated in the VCL component. You need only assign event handlers, set property values, and call methods.

To use the properties, methods, and events of the server object, see the documentation for your server. The component wrapper automatically provides a dual interface where possible. Delphi determines the VTable layout from information in the type library.

In addition, your new component inherits certain important properties and methods from its base class.

ActiveX wrappers

You should always use a component wrapper when hosting ActiveX controls, because the component wrapper integrates the control's window into the VCL framework.

The properties and methods an ActiveX control inherits from TOleControl allow you to access the underlying interface or obtain information about the control. Most applications, however, do not need to use these. Instead, you use the imported control the same way you would use any other VCL control.

Typically, ActiveX controls provide a property page that lets you set their properties. Property pages are similar to the component editors some components display when you double-click on them in the form designer. To display an ActiveX control's property page, right click and choose Properties.

The way you use most imported ActiveX controls is determined by the server application. However, ActiveX controls use a standard set of notifications when they represent the data from a database field. See TOleControl for information on how to host such ActiveX controls.

Automation object wrappers

The wrappers for Automation objects let you control how you want to form the connection to your server object:

First, the ConnectKind property indicates whether the server is local or remote and whether you want to connect to a server that is already running or if a new instance should be launched. When connecting to a remote server, you must specify the machine name using the RemoteMachineName property.

Second, once you have specified the *ConnectKind*, there are three ways you can connect your component to the server:

- you can explicitly connect to the server by calling the component's Connect method.
- You can tell the component to connect automatically when your application starts up by setting the AutoConnect property to *true*.
- You do not need to explicitly connect to the server. The component automatically forms a connection when you use one of the server's properties or methods using the component.

Calling methods or accessing properties is the same as using any other component:

```
[Delphi]
TServerComponent1.DoSomething;
```

```
[C++]
TServerComponent1->DoSomething();
```

Handling events is easy, because you can use the **Object Inspector** to write event handlers. Note, however, that the event handler on your component may have slightly different parameters than those defined for the event in the type library. Specifically, pointer types (var parameters and interface pointers) are changed to Variants. You must explicitly cast var parameters to the underlying type before assigning a value. Interface pointers can be cast to the appropriate interface type using the **as** operator.

For example, the following code shows an event handler for the ExcelApplication event, OnNewWorkBook. The event handler has a parameter that provides the interface of another CoClass (ExcelWorkbook). However, the interface is not passed as an ExcelWorkbook interface pointer, but rather as an OleVariant.

```
[Delphi]
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Note how the OleVariant for the interface must be cast to the correct type }
  ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
```

```
[C++]
void _fastcall TForm1::XLappNewWorkbook(TObject *Sender, ExcelWorkbookPtr Wb)
{
ExcelWorkbook1->ConnectTo(Wb);
}
```

In this example, the event handler assigns the workbook to an ExcelWorkbook component (ExcelWorkbook1). This demonstrates how to connect a component wrapper to an existing interface by using the *ConnectTo* method. The *ConnectTo* method is added to the generated code for the component wrapper.

Servers that have an application object expose a Quit method on that object to let clients terminate the connection. Quit typically exposes functionality that is equivalent to using the File menu to quit the application. Code to call the Quit method is generated in your component's Disconnect method. If it is possible to call the Quit method with no parameters, the component wrapper also has an *AutoQuit* property. *AutoQuit* causes your controller to call Quit when the component is freed. If you want to disconnect at some other time, or if the Quit method requires parameters, you must call it explicitly. Quit appears as a public method on the generated component.

Writing Client Code Based On Type Library Definitions

Although you must use a component wrapper for hosting an ActiveX control, you can write an Automation controller using only the definitions from the type library that appear in the *TypeLibName*_TLB unit. This process is a bit more involved that letting a component do the work, especially if you need to respond to events.

The following topics describe how to implement the various actions your Automation controller needs to perform:

- Connect to the server.
- Control the Automation server using a dual interface.
- Control the Automation server using a dispinterface.
- Respond to events generated by the Automation server.

Connecting to a Server

Before you can drive an Automation server from your controller application, you must obtain a reference to an interface it supports. Typically, you connect to a server through its main interface.

If the main interface is a dual interface, you can use the creator objects in the *TypeLibName*_TLB.pas file. The creator classes have the same name as the CoClass, with the prefix "Co" added. You can connect to a server on the same machine by calling the *Create* method, or a server on a remote machine using the *CreateRemote* method. Because *Create* and *CreateRemote* are class methods, you do not need an instance of the creator class to call them.

```
[Delphi]
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
```

```
[C++]
pInterface = CoServerClassName.Create();
pInterface = CoServerClassName.CreateRemote("Machine1");
```

Create and *CreateRemote* return the default interface for the CoClass.

If the default interface is a dispatch interface, then there is no Creator class generated for the CoClass. Instead, you can call the global CreateOleObject function, passing in the GUID for the CoClass (there is a constant for this GUID defined at the top of the _TLB unit). *CreateOleObject* returns an IDispatch pointer for the default interface.

Controlling an Automation Server Using a Dual Interface

After using the automatically generated creator class to connect to the server, you call methods of the interface. For example,

```
[Delphi]
var
  MyInterface : _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
```

```
[C++]
TComApplication AppPtr = CoWordApplication_.Create();
AppPtr->DoSomething;
```

The interface and creator class are defined in the *TypeLibName*_TLB unit that is generated automatically when you import a type library.

Controlling an Automation Server Using a Dispatch Interface

Typically, you use the dual interface to control the Automation server. However, you may find a need to control an Automation server with a dispatch interface because no dual interface is available.

To call the methods of a dispatch interface,

1. Connect to the server, using the global CreateOleObject function.
2. Use the **as** operator to cast the *IDispatch* interface returned by *CreateOleObject* to the dispinterface for the CoClass. This dispinterface type is declared in the *TypeLibName*_TLB unit.
3. Control the Automation server by calling methods of the dispinterface.

   Another way to use dispatch interfaces is to assign them to a *Variant*. By assigning the interface returned by *CreateOleObject* to a *Variant*, you can take advantage of the *Variant* type's built-in support for interfaces. Simply call the methods of the interface, and the *Variant* automatically handles all *IDispatch* calls, fetching the dispatch ID and invoking the appropriate method. The Variant type includes built-in support for calling dispatch interfaces, through its **var.**

```
  V: Variant;
begin
  V:= CreateOleObject("TheServerObject");
  V.MethodName; { calls the specified method }
  ...
```

An advantage of using *Variants* is that you do not need to import the type library, because *Variants* use only the standard *IDispatch* methods to call the server. The trade-off is that *Variants* are slower, because they use dynamic binding at runtime.

Handling Events in an Automation Controller

When you generate a Component wrapper for an object whose type library you import, you can respond to events simply using the events that are added to the generated component. If you do not use a Component wrapper, however, (or if the server uses COM+ events), you must write the event sink code yourself.

Handling Automation events programmatically

Before you can handle events, you must define an event sink. This is a class that implements the event dispatch interface that is defined in the server's type library.

To write the event sink, create an object that implements the event dispatch interface:

```
[Delphi]
TServerEventsSink = class(TObject, _TheServerEvents)
...{ declare the methods of _TheServerEvents here }
end;
```

```
[C++]
class MyEventSinkClass: TEventDispatcher<MyEventSinkClass, DIID_TheServerEvents>
{
...// declare the methods of DIID_TheServerEvents here
}
```

Once you have an instance of your event sink, you must inform the server object of its existence so that the server can call it. To do this, you call the global InterfaceConnect procedure, passing it

- The interface to the server that generates events.
- The GUID for the event interface that your event sink handles.
- An IUnknown interface for your event sink.
- A variable that receives a Longint that represents the connection between the server and your event sink.

```
[Delphi]
{MyInterface is the server interface you got when you connected to the server }
InterfaceConnect(MyInterface, DIID_TheServerEvents,
                 MyEventSinkObject as IUnknown, cookievar);
```

```
[C++]
pInterface = CoServerClassName.CreateRemote("Machine1");
MyEventSinkClass ES;
ES.ConnectEvents(pInterface);
```

After calling *InterfaceConnect*, your event sink is connected and receives calls from the server when events occur.

You must terminate the connection before you free your event sink. To do this, call the global InterfaceDisconnect procedure, passing it all the same parameters except for the interface to your event sink (and the final parameter is ingoing rather than outgoing):

```
[Delphi]

InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
```

```
[C++]
ES.DisconnectEvents(pInterface);
```

Note: You must be certain that the server has released its connection to your event sink before you free it. Because you don't know how the server responds to the disconnect notification initiated by *InterfaceDisconnect*, this may lead to a race condition if you free your event sink immediately after the call. The easiest way to guard against problems is to have your event sink maintain its own reference count that is not decremented until the server releases the event sink's interface.

Handling COM+ events

Under COM+, servers use a special helper object to generate events rather than a set of special interfaces (*IConnectionPointContainer* and *IConnectionPoint*). Because of this, you can't use an event sink that descends from *TEventDispatcher. TEventDispatcher* is designed to work with those interfaces, not COM+ event objects.

Instead of defining an event sink, your client application defines a subscriber object. Subscriber objects, like event sinks, provide the implementation of the event interface. They differ from event sinks in that they subscribe to a particular event object rather than connecting to a server's connection point.

To define a subscriber object, use the COM Object wizard, selecting the event object's interface as the one you want to implement. The wizard generates an implementation unit with skeletal methods that you can fill in to create your event handlers.

Note: You may need to add the event object's interface to the registry using the wizard if it does not appear in the list of interfaces you can implement.

Once you create the subscriber object, you must subscribe to the event object's interface or to individual methods (events) on that interface. There are three types of subscriptions from which you can choose:

- **Transient subscriptions.** Like traditional event sinks, transient subscriptions are tied to the lifetime of an object instance. When the subscriber object is freed, the subscription ends and COM+ no longer forwards events to it.
- **Persistent subscriptions**. These are tied to the object class rather than a specific object instance. When the event occurs, COM locates or launches an instance of the subscriber object and calls its event handler. In-process objects (DLLs) use this type of subscription.
- **Per-user subscriptions**. These subscriptions provide a more secure version of transient subscriptions. Both the subscriber object and the server object that fires events must be running under the same user account on the same machine.

Note: Objects that subscribe to COM+ events must be installed in a COM+ application.

Creating Clients for Servers That Do Not Have a Type Library

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. To write clients that host such objects, you can use the TOleContainer component. This component appears on the System category of the **Tool Palette.**

TOleContainer acts as a host site for an Ole2 object. It implements the *IOleClientSite* interface and, optionally, *IOleDocumentSite*. Communication is handled using OLE verbs.

To use TOleContainer

1. Place a *TOleContainer* component on your form.
2. Set the *AllowActiveDoc* property to *true* if you want to host an Active document.
3. Set the *AllowInPlace* property to indicate whether the hosted object should appear in the *TOleContainer*, or in a separate window.
4. Write event handlers to respond when the object is activated, deactivated, moved, or resized.
5. To bind the *TOleContainer* object at design time, right click and choose Insert Object. In the Insert Object dialog, choose a server object to host.
6. To bind the *TOleContainer* object at runtime, you have several methods to choose from, depending on how you want to identify the server object. These include CreateObject, which takes a program id, CreateObjectFromFile, which takes the name of a file to which the object has been saved, CreateObjectFromInfo, which takes a record containing information on how to create the object, or CreateLinkToFile, which takes the name of a file to which the object was saved and links to it rather than embeds it.
7. Once the object is bound, you can access its interface using the OleObjectInterface property. However, because communication with Ole2 objects was based on OLE verbs, you will most likely want to send commands to the server using the DoVerb method.
8. When you want to release the server object, call the DestroyObject method.

Using .NET Assemblies with Delphi

The Microsoft .NET Framework and the Common Language Runtime (CLR) provide a runtime environment in which components written in .NET languages can seamlessly interact with each other. A compiler for a .NET language does not emit native machine code. Instead, the language is compiled to an intermediate, platform neutral form called Microsoft Intermediate Language (MSIL, or IL for short). The modules containing IL code are linked together to form an assembly. An assembly can be made up of multiple modules, or it can be a single file. In either case, an assembly is a self-describing entity; it holds information about the types it contains, the modules that comprise the assembly, and dependencies on other assemblies. An assembly is the basic unit of deployment in the .NET development environment, and the CLR manages loading, compilation to native machine code, and subsequent execution of that code. Applications that run entirely within the context of the CLR are called **managed code**.

One of the services provided by the CLR is the ability for managed code to call on **unmanaged code**, that is, code that was compiled to native machine language and which does not execute within the environment of the CLR. For example, through a service called Platform Invoke (often shortened to PInvoke), managed code can call on native Win32 APIs. This ability extends to using legacy COM objects from a managed .NET application. The ability to interoperate between managed code and COM objects also goes in the other direction, making it possible to expose .NET components to unmanaged applications. To the unmanaged application, loading and accessing the .NET component almost entirely the same as accessing any other COM object.

Requirements for COM Interoperability

If you are developing new components with the .NET Framework, then you need to install the full .NET Framework SDK, which is available from Microsoft's MSDN website: **msdn.microsoft.com**. If you are only using .NET types

directly from the .NET Framework core assemblies, then you only need to install the .NET Framework Redistributable, also available from the MSDN website. Of course, any unmanaged application that relies on services provided by the .NET Framework will require the .NET Framework Redistributable to be deployed on the end-user's machine.

.NET components are exposed to unmanaged code through the use of proxy objects called COM Callable Wrappers (CCW). Since COM mechanisms are used to make the bridge between unmanaged and managed code, you must register the .NET assemblies that contain components you wish to use. Use the .NET Framework utiltity called **regasm** to create the necessary registry entries. The process is similar to registering any other COM object, and will be covered in more detail later in this section.

The .NET assembly **mscorlib.dll** contains the types that are integral to the .NET Framework. All .NET assemblies must reference the mscorlib assembly, simply because it provides the core functionality of the .NET Framework on the Microsoft Windows platform. If you will be using types directly contained in the mscorlib assembly, then you must run the regasm utility on mscorlib.dll. The Delphi installer registers the mscorlib assembly for you, if it is not already registered.

.NET components can be deployed in two ways: In a global, shared location called the Global Assembly Cache (GAC), or together in the same directory as the executable. Components that are shared among multiple applications should be deployed in the GAC. Because they are shared, and because of the side-by-side deployment capabilities of the .NET Framework, assemblies deployed in the GAC must be given a strong name (i.e. they must be digitally signed). The .NET Framework contains a utility called sn, which is used to generate the encryption keys. After the keys have been generated and the component has been built, the assembly is installed into the global assembly cache using another .NET utility called **gacutil**.

A .NET component can also be deployed in the same directory as the unmanaged executable. In this deployment scenario, the strong key and GAC installation utility are not required. However, the component must still be registered using the regasm utility. Unlike an ordinary COM object, registering a .NET component does not make it accessible to an application outside of the directory where the component is deployed.

.NET Components and Type Libraries

Both COM, and the .NET Framework contain mechanisms to expose type information. In COM, one such mechansim is the type library. Type libraries are a binary, programming language-neutral way for a COM object to expose type metadata at runtime. Because type libraries are opened and parsed by system APIs, languages such as Delphi can import them and gain the advantages of vtable binding, even if the component was written in a different programming language.

In the .NET development environment, the assembly doubles as a container for both IL, and type information. The .NET Framework contains classes that are used to examine (or, "reflect") the types contained in an assembly. When you access a .NET component from unmanaged code, you are actually using a proxy (the COM Callable Wrapper, mentioned earlier), not the .NET component itself. The CCW mechanism, plus the self-describing nature of assemblies, is enough to allow you to access a .NET component entirely through late binding.

Because you can access a .NET component through late binding, creating a type library for the component is not strictly required. All that is required is that the assembly be registered. In fact, unmanaged clients are restricted to late binding by default. Depending on how the .NET component was designed and built, you might find only an "empty" class interface if you inspect its type library. Such a type library is useless, in terms of enabling clients to use vtable binding instead of late binding through *IDispatch*.

The following example demonstrates how to late bind to the ArrayList collection class contained in mscorlib.dll. The mscorlib assembly must be registered prior to using any type in the manner described here. The Delphi installer automatically registers mscorlib, but you can run the regasm utility again if need be (e.g. you unregistered mscorlib with the /u regasm option). Execute the command

```
regasm mscorlib.dll
```

in the .NET Framework directory to register the mscorlib assembly.

Note: Do not use the **/tlb** option when registering mscorlib.dll. The .NET Framework already includes a type library for the mscorlib assembly; you do not need to create a new one.

The following code is attached to a button click event of a Delphi form:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   capacity: Integer;
   item:Variant;
   dotNetArrayList:Variant;
begin
   { Create the object }
   dotNetArrayList := CreateOleObject('System.Collections.ArrayList');

   { Get the capacity of the ArrayList }
   capacity := dotNetArrayList.Capacity;

   { Add an item }
   dotNetArrayList.Add('A string item');

   { Retrieve the item, using the Array interface method, Item(). }
   item := dotNetArrayList.Item(0);

   {Remove all items }
   dotNetArrayList.Clear;
end;
```

Accessing User-defined .NET Components

When you examine a type library for a .NET component, you might - depending on how the component was designed and built - find only an empty class interface. The class interface will not contain any information about the parameters expected by the methods implemented by the class. Also notably absent, are the dispids for the methods of the class. The reason for this are the problems that can arise when a new version of the component is created.

In COM, inheriting via interface is the only option. In the .NET Framework, inheriting via interface or inheriting via implementation is a design decision. .NET component writers can choose to add a new method or property at any time. If changes are made to the .NET component, any COM client that depends on the layout of the interface (e.g. by caching dispids) will break.

A .NET component writer must choose to expose type information in an exported type library; it is not the default behavior. This is done through the use of the **ClassInterfaceAttribute** custom attribute. ClassInterfaceAttribute is found in the **System.Runtime.InteropServices** namespace. It can take on the values of the **ClassInterfaceType** enumeration, which are, **AutoDispatch** (the default), **AutoDual**, and **None**.

The AutoDispatch value is what causes the empty class interface to be generated. Clients are restricted to late binding when accessing such a class. The AutoDual value causes all type information (including dispids) to be included for a class so marked. When a class is marked with the AutoDual value, type information is also included for all inherited classes. This is the most convenient approach, and it can work well when the .NET components are developed in a controlled environment. However, this approach is also the one most prone to the versioning problems mentioned earlier.

The ClassInterfaceType value None inhibits the generation of a class interface. When a .NET class is marked this way, only the methods implemented in inherited interfaces can be invoked. For .NET components that are intended to be used by an unmanaged COM client, inheritance via interface is the preferred method of interoperating between managed and unmanaged code. This way, the COM client is less susceptible to changes in the .NET class. It also reinforces a tried-and-true COM design principle, the immutability of interfaces.

The following example demonstrates this approach. We start out with a C# interface called MyInterface, and a class called MyClass.

```
   using System;
   using System.Reflection;
   using System.Runtime.InteropServices;
   using System.Windows.Forms;

   [assembly:AssemblyKeyFile("KeyFile.snk")]
   namespace InteropTest1 {
      public interface MyInterface {
         void myMethod1();
         void myMethod2(string msg);
      }

   // Restrict clients to using only implemented interfaces.
   [ClassInterface(ClassInterfaceType.None)]
   public class MyClass : MyInterface {

      // The class must have a parameterless constructor for COM interoperability
      public MyClass() {
      }

      // Implement MyInterface methods
      public void myMethod1() {
         MessageBox.Show("In C# Method!");
      }

      public void myMethod2(string msg) {
         MessageBox.Show(msg);
      }
   }
}
```

The assembly is marked with the **AssemblyKeyFile** attribute. This is required if the component is to be deployed in the Global Assembly Cache. If you deploy your component in the same directory as the unmanaged executable client, the strong key is not required. This example component will be deployed in the GAC, so we first generate the keyfile using the Strong Name Utility of the .NET Framework SDK:

```
sn -k KeyFile.snk
```

Execute this command from the same directory where the C# source file is located.

The next step is to compile this code using the C# compiler. Assuming the C# code is in a file called interoptest1.cs:

```
csc /t:library interoptest1.cs
```

The result of this command is the creation of an assembly called interoptest1.dll. The assembly must now be registered, using the regasm utility. Regasm is similar in concept to tregsvr; it creates entries in the Windows registry that allow the component to be exposed to unmanaged COM clients.

```
regasm /tlb interoptest1.dll
```

The use of the **/tlb** option causes regasm to do two things: First, the registry entries for the assembly are created. Second, the types in the assembly will be exported to a type library, and the type library will also be registered.

Finally, the component is deployed to the GAC using the gacutil command:

```
gacutil -i interoptest1.dll
```

The **-i** option indicates the assembly is being installed into the GAC. The gacutil command must be executed each time you build a new version of the .NET component. Later, if you wish to remove the component from the GAC, execute the gacutil command again, this time with the **-u** option:

```
gacutil -u interoptest1
```

Note: When uninstalling a component, do not include the '.dll' extension on the assembly name.

Once the .NET component has been built, registered, and installed into the GAC (or, copied to the directory of the unmanaged executable), accessing it in Delphi is the same as for any other COM object. Open or create your project, and then select **Component ▸ Import type library** from the menu. Scroll through the list of registered type libraries until you find the one for your component. You can create a package for the component and install it on the **Tool Palette** by selecting the Install check box. The type library importer will create a _TLB file to wrap the component, making it accessible to unmanaged Delphi code through vtable binding.

The Add button of the type library import dialog box will not correctly register a type library exported for a .NET assembly. Instead, you must always use the regasm utility on the command line.

The type library importer will automatically create _TLB files (and their corresponding .dcr and .dcu files) for any .NET assemblies that are referenced in the imported type library. Importing the type library for the example C# component above would cause the creation of _TLB, .dcr, and .dcu files for the mscorlib and System.Windows.Forms assemblies.

The example below demonstrates calling methods on the .NET component, after its type library has been imported into Delphi. The class and method names come from the earlier C# example, and the variable MyClass1 is assumed to be previously declared (e.g. as a member variable of a class, or a local variable of a procedure or function).

```
MyClass1 := TMyClass.Create(self);
MyClass1.myMethod1;
MyClass1.myMethod2('Display this message');
MyClass1.Free;
```

Creating simple COM servers

Creating Simple COM Servers: Overview

Delphi provides wizards to help you create various COM objects. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call.

Two wizards, in particular, ease the process of creating simple COM objects:

- The COM Object wizard builds a lightweight COM object whose default interface descends from IUnknown or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.
- The Automation Object wizard creates a simple Automation object whose default interface descends from IDispatch. *IDispatch* introduces a standard marshaling mechanism and support for late binding of interface calls.

Note: COM defines many standard interfaces and mechanisms for handling specific situations. The Delphi wizards automate the most common tasks. However, some tasks, such as custom marshaling, are not supported by any Delphi wizards. For information on that and other technologies not explicitly supported by Delphi, refer to the Microsoft Developer's Network (MSDN) documentation. The Microsoft Web site also provides current information on COM support.

Overview of creating a COM object

Whether you use the **Automation Object wizard** to create a new Automation server or the COM object wizard to create some other type of COM object, the process you follow is the same.

It involves these steps:

1. Design the COM object.
2. Use the COM Object wizard or the Automation Object wizard to create the server object.
3. Define the interface that the object exposes to clients.
4. Register the COM object.
5. Test and debug the application.

Designing a COM Object

When designing the COM object, you need to decide what COM interfaces you want to implement. You can write a COM object to implement an interface that has already been defined, or you can define a new interface for your object to implement. In addition, you can have your object support more than one interface. For information about standard COM interfaces that you might want to support, see the MSDN documentation.

- To create a COM object that implements an existing interface, use the COM Object wizard.
- To create a COM object that implements a new interface that you define, use either the COM Object wizard or the Automation Object wizard. The COM object wizard can generate a new default interface that descends from IUnknown, and the Automation object gives your object a default interface that descends from IDispatch. No matter which wizard you use, you can always use the Type Library editor later to change the parent interface of the default interface that the wizard generates.

In addition to deciding what interfaces to support, you must decide whether the COM object is an in-process server, out-of-process server, or remote server. For in-process servers and for out-of-process and remote servers that use a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers.

Using the COM Object Wizard

The COM object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from TCOMObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards.
- Optionally, adds a type library to your project and adds your object and its interface to the type library.

Before you create a COM object, create or open the project for the application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard

1. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
2. Select the folder labeled **ActiveX** under **Delphi Projects**
3. Double-click the COM object icon in the right pane.

In the wizard, you must specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. The class created to implement your object has this name with a 'T' prepended. If you do not choose to implement an existing interface, the wizard gives your CoClass a default interface that has this name with an 'I' prepended.
- **Implemented Interface:** By default, the wizard gives your object a default interface that descends from IUnknown. After exiting the wizard, you can then use the Type Library editor to add properties and methods to this interface. However, you can also select a pre-defined interface for your object to implement. Click the List button in the COM object wizard to bring up the Interface Selection wizard, where you can select any dual or custom interface defined in a type library registered on your system. The interface you select becomes the default interface for your new CoClass. The wizard adds all the methods on this interface to the generated implementation class, so that you only need to fill in the bodies of the methods in the implementation unit. Note that if you select an existing interface, the interface is not added to your project's type library. This means that when deploying your object, you must also deploy the type library that defines the interface.

- Instancing: Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them.
- Threading Model: Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on how to provide thread support to your application, see Writing multi-threaded applications
- Include Type Library: You can choose whether you want to include a type library for your object. This is recommended for two reasons: it lets you use the Type Library editor to define interfaces, thereby updating much of the implementation, and it gives clients an easy way to obtain information about your object and its interfaces. If you are implementing an existing interface, Delphi requires your project to use a type library. This is the only way to provide access to the original interface declaration.
- Mark interfaceOleautomation: If you have opted to create a type library and are willing to confine yourself to Automation-compatible types, you can let COM handle the marshaling for you when you are not generating an in-process server. By marking your object's interface as OleAutomation in the type library, you enable COM to set up the proxies and stubs for you and handles passing parameters across process boundaries. You can only specify whether your interface is Automation-compatible if you are generating a new interface. If you select an existing interface, its attributes are already specified in its type library. If your object's interface is not marked as OleAutomation, you must either create an in-process server or write your own marshaling code.

You can optionally add a description of your COM object. This description appears in the type library for your object if you create one.

Using the Automation Object Wizard

The Automation object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from TAutoObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards.
- Adds a type library to your project and adds your object and its interface to the type library.

Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

To display the Automation wizard:

1. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
2. Select the folder labeled **ActiveX** under **Delphi Projects**.
3. Double-click the **Automation Object** icon in the right pane.
4. In the wizard dialog, specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. Your object's default interface is created with a name based on this CoClass name with an 'I' prepended, and the class created to implement your object has this name with a 'T' prepended.
- Instancing: Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them.
- Threading Model: Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how

the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on how to provide thread support to your application, see Writing multi-threaded applications.

- Generate Event support code: You must indicate whether you want your object to generate events to which clients can respond. The wizard can provide support for the interfaces required to generate events and the dispatching of calls to client event handlers.

The Automation object implements a dual interface, which supports both early (compile-time) binding through the VTable and late (runtime) binding through the IDispatch interface.

COM Object Instancing Types

Many of the COM wizards require you to specify an instancing mode for the object. Instancing determines how many instances of your object clients can create in a single executable. If you specify a Single Instance model, for example, then once a client has instantiated your object, COM removes the application from view so that other clients must launch their own instances of the application. Because this affects the visibility of your application as a whole, the instancing mode must be consistent across all objects in your application that can be instantiated by clients. That is, you should not create one object in your application that uses Single Instance mode and another in the same application that uses Multiple Instance mode.

Note: Instancing is ignored when your COM object is used only as an in-process server.

When the wizard creates a new COM object, it can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly, although your application can create the object and pass an interface for it to clients.
Single Instance	Allows clients to create only a single instance of the object for each executable (application), so creating multiple instances results in launching multiple instances of the application. Each client has its own dedicated instance of the server application.
Multiple Instances	Specifies that multiple clients can create instances of the object in the same process space.

Choosing a Threading Model

When creating an object using a wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance, because multiple clients can access your application at the same time.

The following table lists the different threading models you can specify.

Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	The server provides no thread support. COM serializes client requests so that the application receives one request at a time.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	COM ensures that only one client thread can call the object at a time. All client calls use the thread in which the object was created.	Objects can safely access their own instance data, but global data must be protected using critical sections or some other form of serialization. The thread's local variables are reliable across multiple calls.

		Some performance benefits.
Free (also called multi-threaded apartment)	Objects can receive calls on any number of threads at any time.	Objects must protect all instance and global data using critical sections or some other form of serialization. Thread local variables are not reliable across multiple calls.
Both	This is the same as the Free-threaded model except that outgoing calls (for example, callbacks) are guaranteed to execute in the same thread.	Maximum performance and flexibility. Does not require the application to provide thread support for parameters supplied to outgoing calls.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict.	You must guard against thread conflicts involving global data and any instance data that is accessed by multiple methods. This model should not be used with objects that have a user interface (visual controls). This model is only available under COM+. Under COM, it is mapped to the Apartment model.

Note: Local variables (except those in callbacks) are always safe, regardless of the threading model. This is because local variables are stored on the stack and each thread has its own stack. Local variables may not be safe in callbacks when using free-threading.

The threading model you choose in the wizard determines how the object is registered in the system Registry. You must make sure that your object implementation adheres to the threading model you have chosen. For general information on writing thread-safe code, see Writing multi-threaded applications.

For in-process servers, setting the threading model in the wizard sets the threading model key in the CLSID registry entry.

Out-of-process servers are registered as EXE, and Delphi initializes COM for the highest threading model required. For example, if an EXE includes a free-threaded object, it is initialized for free threading, which means that it can provide the expected support for any free-threaded or apartment-threaded objects contained in the EXE. To manually override threading behavior in EXEs, use the CoInitFlags variable.

Writing an object that supports the free threading model

Use the free threading (or both) model rather than apartment threading whenever the object needs to be accessed from more than one thread. A common example is a client application connected to an object on a remote machine. When the remote client calls a method on that object, the server receives the call on a thread from the thread pool on the server machine. This receiving thread makes the call locally to the actual object; and, because the object supports the free threading model, the thread can make a direct call into the object.

If the object supported the apartment threading model instead, the call would have to be transferred to the thread on which the object was created, and the result would have to be transferred back into the receiving thread before returning to the client. This approach requires extra marshaling.

To support free threading, you must consider how instance data can be accessed for *each* method. If the method is writing to instance data, you must use critical sections or some other form of serialization, to protect the instance data. Likely, the overhead of serializing critical calls is less than executing COM's marshaling code.

Note that if the instance data is read-only, serialization is not needed.

Free-threaded in-process servers can improve performance by acting as the outer object in an aggregation with the free-threaded marshaler. The free-threaded marshaler provides a shortcut for COM's standard thread handling when a free-threaded DLL is called by a host (client) that is not free-threaded.

To aggregate with the free threaded marshaler, you must

- Call *CoCreateFreeThreadedMarshaler*, passing your object's *IUnknown* interface for the resulting free-threaded marshaler to use: `CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);` `CoCreateFreeThreadedMarshaler(static_cast<IUnknown *>(this), &FMarshaler);`. This line assigns the interface for the free-threaded marshaler to a class member, *FMarshaler*.
- Using the **Type Library Editor**, add the *IMarshal* interface to the set of interfaces your CoClass implements.
- In your object's *QueryInterface* method, delegate calls for IDD_IMarshal to the free-threaded marshaler (stored as *FMarshaler* above).

Warning: The free-threaded marshaler violates the normal rules of COM marshaling to provide additional efficiency. It should be used with care. In particular, it should only be aggregated with free-threaded objects in an in-process server, and should only be instantiated by the object that uses it (not another thread).

Writing an object that supports the apartment threading model

To implement the (single-threaded) apartment threading model, you must follow a few rules:

- The first thread in the application that gets created is COM's main thread. This is typically the thread on which WinMain was called. This must also be the last thread to uninitialize COM.
- Each thread in the apartment threading model must have a message loop, and the message queue must be checked frequently.
- When a thread gets a pointer to a COM interface, that pointer may only be used in that thread.

The single-threaded apartment model is the middle ground between providing no threading support and full, multi-threading support of the free threading model. A server committing to the apartment model promises that the server has serialized access to all of its global data (such as its object count). This is because different objects may try to access the global data from different threads. However, the object's instance data is safe because the methods are always called on the same thread.

Typically, controls for use in Web browsers use the apartment threading model because browser applications always initialize their threads as apartment.

Writing an object that supports the neutral threading model

Under COM+, you can use another threading model that is in between free threading and apartment threading: the neutral model. Like the free-threading model, this model allows multiple threads to access your object at the same time. There is no extra marshaling to transfer to the thread on which the object was created. However, your object is guaranteed to receive no conflicting calls.

Writing an object that uses the neutral threading model follows much the same rules as writing an apartment-threaded object, except that you do need to guard instance data against thread conflicts if it can be accessed by different methods in the object's interface. Any instance data that is only accessed by a single interface method is automatically thread-safe.

Defining a COM Object's Interface

When you use a wizard to create a COM object, the wizard automatically generates a type library (unless you specify otherwise in the COM object wizard). The type library provides a way for host applications to find out what the object

can do. It also lets you define your object's interface using the Type Library editor. The interfaces you define in the Type Library editor define what properties, methods, and events your object exposes to clients.

Note: If you selected an existing interface in the COM object wizard, you do not need to add properties and methods. The definition of the interface is imported from the type library in which it was defined. Instead, simply locate the methods of the imported interface in the implementation unit and fill in their bodies.

Adding a property to the object's interface

When you add a property to your object's interface using the **Type Library Editor**, it automatically adds a method to read the property's value and/or a method to set the property's value. The **Type Library Editor**, in turn, adds these methods to your implementation class, and in your implementation unit creates empty method implementations for you to complete.

To add a property to your object's interface

1. In the **Type Library Editor**, select the default interface for the object.

   The default interface should be the name of the object preceded by the letter "I." To determine the default, in the **Type Library Editor**, click the CoClass and then select the **Implements** tab, and check the list of implemented interfaces for the one marked, "Default."
2. To expose a read/write property, click the **New Property** button on the toolbar; otherwise, click the arrow next to the **New Property** button on the toolbar, and then click the type of property to expose.
3. In the **Attributes** pane, specify the name and type of the property.
4. On the **Type Library Editor** toolbar, click the **Refresh Implementation** button.

   A definition and skeletal implementations for the property access methods are inserted into the object's implementation unit.
5. In the implementation unit, locate the access methods for the property. These have names of the form `Get_PropertyName` and `Set_PropertyName`. Add code that gets or sets the property value of your object. This code may simply call an existing function inside the application, access a data member that you add to the object definition, or otherwise implement the property.

Adding a method to the object's interface

When you add a method to your object's interface using the **Type Library Editor**, the **Type Library Editor** can, in turn, add the methods to your implementation class, and in your implementation unit create empty implementation for you to complete.

To expose a method via your object's interface

1. In the **Type Library Editor**, select the default interface for the object.

   The default interface should be the name of the object preceded by the letter "I". To determine the default, in the **Type Library Editor**, click the CoClass and select the **Implements** tab, and check the list of implemented interfaces for the one marked, "Default."
2. Click the **New Method** button.
3. In the **Attributes** pane, specify the name of the method.
4. In the **Parameters** pane, specify the method's return type and add the appropriate parameters.
5. On the **Type Library Editor** toolbar, click the **Refresh Implementation** button.

A definition and skeletal implementation for the method is inserted into the object's implementation unit.

6 In the implementation unit, locate the newly inserted method implementation. The method is completely empty. Fill in the body to perform whatever task the method represents.

Exposing events to clients

There are two types of events that a COM object can generate: traditional events and COM+ events.

- COM+ events require that you create a separate event object using the event object wizard and add code to call that event object from your server object.
- You can use the wizard to handle much of the work in generating traditional events. This process is described below.

Note: The COM object wizard does not generate event support code. If you want your object to generate traditional events, you should use the **Automation object wizard**.

In order for an object to generate events, you need to do the following:

1 In the **Automation Object wizard**, check the box, **Generate event support code**.

The wizard creates an object that includes an Events interface as well as the default interface. This Events interface has a name of the form `I` *CoClassname* `Events`. It is an outgoing (source) interface, which means that it is not an interface your object implements, but rather is an interface that clients must implement and which your object calls. (You can see this by selecting your CoClass, going to the **Implements** page, and noting that the **Source** column on the Events interface says true.)

In addition to the Events interface, the wizard adds the *IConnectionPointContainer* interface to the declaration of your implementation class, and adds several class members for handling events. Of these new class members, the most important are *FConnectionPoint* and *FConnectionPoints*, which implement the *IConnectionPoint* and *IConnectionPointContainer* interfaces using built-in VCL classes. *FConnectionPoint* is maintained by another method that the wizard adds, *EventSinkChanged*.

2 In the **Type Library Editor**, select the outgoing Events interface for your object. (This is the one with a name of the form `I` *CoClassName* `Events`)

3 Click the **New Method** button from the **Type Library Editor** toolbar. Each method you add to the Events interface represents an event handler that the client must implement.

4 In the **Attributes** pane, specify the name of the event handler, such as `MyEvent`.

5 On the **Type Library Editor** toolbar, click the **Refresh Implementation** button.

Your object implementation now has everything it needs to accept client event sinks and maintain a list of interfaces to call when the event occurs. To call these interfaces, you can create a method to generate each event on clients.

6 In the **Code Editor**, add a method to your object for firing each event. For example,

```
[Delphi]

unit ev;
interface
uses
   ComObj, AxCtrls, ActiveX, Project1_TLB;
type
   TMyAutoObject = class (TAutoObject,IConnectionPointContainer, IMyAutoObject)
```

```
private
   .
   .
   .
public
   procedure Initialize; override;
   procedure Fire_MyEvent; { Add a method to fire the event}
```

7 Implement the method you added in the last step so that it iterates through all the event sinks maintained by your object's *FConnectionPoint* member:

```
[Delphi]

procedure TMyAutoObject.Fire_MyEvent;
var
  I: Integer;
  EventSinkList: TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList :=FConnectionPoint.SinkList; {get the list of client sinks }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;
```

8 Whenever you need to fire the event so that clients are informed of its occurrence, call the method that dispatches the event to all event sinks:

```
[Delphi]
if EventOccurs then Fire_MyEvent; { Call method you created to fire events.}
```

```
[C++]
if (EventOccurs) Fire_MyEvent; // Call method you created to fire events.
```

Managing Events in Your Automation Object

The Automation wizard automatically generates event code if you check the option, Generate Support Code in the Automation Object wizard dialog box.

For a server to support traditional COM events, it must provide the definition of an outgoing interface which is implemented by a client. This outgoing interface includes all the event handlers the client must implement to respond to server events.

When a client has implemented the outgoing event interface, it registers its interest in receiving event notification by querying the server's *IConnectionPointContainer* interface. The *IConnectionPointContainer* interface returns the server's *IConnectionPoint* interface, which the client then uses to pass the server a pointer to its implementation of the event handlers (known as a sink).

The server maintains a list of all client sinks and calls methods on them when an event occurs.

When you select Generate Event Support Code, Delphi automatically generates the code necessary to support IConnectionPoint and IConnectionPointContainer. This support, and the way you can use it to generate events is described in Exposing events to clients.

Automation Interfaces

The **Automation Object** wizard implements a dual interface by default, which means that the Automation object supports both

- Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or dispinterface.
- Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a custom interface.

Note: Any interfaces generated by the **COM Object** wizard that do not descend from *IDispatch* only support VTable calls.

Dual Interfaces

A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch. For* those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used.

Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- For VTable interfaces, the compiler performs type checking and provides more informative error messages.
- For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object.
- For in-process servers, you have the benefit of fast access through VTable interfaces.
- For out-of-process servers, COM marshals data for both VTable interfaces and dispinterfaces. COM provides a generic proxy/stub implementation that can marshal the interface based on the information contained in a type library.

The first three entries of the VTable for a dual interface refer to the *IUnknown* interface, the next four entries refer to the *IDispatch* interface, and the remaining entries are COM entries for direct access to members of the custom interface.

Dispatch Interfaces

Automation controllers are clients that use the COM *IDispatch* interface to access the COM server objects. The controller must first create the object, then query the object's *IUnknown* interface for a pointer to its *IDispatch* interface. *IDispatch* keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through *IDispatch*, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the *IDispatch* method, *GetIDsOfNames*.

Once it has the dispID, the controller can then call the *IDispatch* method, *Invoke,* to execute the appropriate code (property or method), packaging the parameters for the property or method into one of the *Invoke* parameters. *Invoke* has a fixed compile-time signature that allows it to accept any number of arguments when calling an interface method.

The Automation object's implementation of *Invoke* must then unpackage the parameters, call the property or method, and be prepared to handle any errors that occur. When the property or method returns, the object passes its return value back to the controller.

This is called late binding because the controller binds to the property or method at runtime rather than at compile time.

Custom Interfaces

Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*.

If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the **Type Library Editor**. If the object has a type library and also supports *IDispatch*, a client can also get the dispIDs of the *IDispatch* interface and bind directly to a VTable offset. Delphi's type library importer (TLIBIMP) retrieves dispIDs at import time, so clients that use dispinterfaces can avoid calls to *GetIDsOfNames; this* information is already in the _TLB unit. However, clients still need to call *Invoke*.

Marshaling Data

For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling:

- Automatically, using the *IDispatch* interface.
- Automatically, by creating a type library with your server and marking the interface with the OLE Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling.
- Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**.

Note: The first method (using *IDispatch*) is only available on Automation servers. The second method is automatically available on all objects that are created by wizards and which use a type library.

Automation compatible types

Function result and parameter types of the methods declared in dual and dispatch interfaces and interfaces that you mark as OLE Automation must be *Automation-compatible* types. The following types are OLE Automation-compatible:

First, the predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see Valid types.

Second, enumeration types defined in a type library. OLE Automation-compatible enumeration types are stored as 32-bit values and are treated as values of type *Integer* for purposes of parameter passing.

Third, interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.

Fourth, dispinterface types defined in a type library.

Fifth, any custom record type defined within the type library.

Sixth, *IFont*, *IStrings*, and *IPicture*. Helper objects must be instantiated to map

- an *IFont to* a *TFont*

- an *IStrings* to a *TStrings*
- an *IPicture* to a *TPicture*

The ActiveX control and ActiveForm wizards create these helper objects automatically when needed. To use the helper objects, call the global routines, GetOleFont, GetOleStrings, GetOlePicture, respectively.

Type restrictions for automatic marshaling

For an interface to support automatic marshaling (also called Automation marshaling or type library marshaling), the following restrictions apply. When you edit your object using the type library editor, the editor enforces these restrictions:

- String data types must be transferred as wide strings (BSTR). PChar and AnsiString cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function's return value. If the method is declared using the safecall calling convention, this condition is imposed automatically, with the declared return type converted to an output parameter.
- Members of a dual interface that need to return other values should specify these parameters as **var** or **out**, indicating an output parameter that returns the value of the function.

Note: One way to bypass the Automation types restrictions is to implement a separate *IDispatch* interface and a custom interface. By doing so, you can use the full range of possible argument types. This means that COM clients have the option of using the custom interface, which Automation controllers can still access. In this case, though, you must implement the marshaling code manually.

Custom marshaling

Typically, you use automatic marshaling in out-of-process and remote servers because it is easier—COM does the work for you. However, you may decide to provide custom marshaling if you think you can improve marshaling performance. When implementing your own custom marshaling, you must support the *IMarshal* interface. For more information, on this approach, see the Microsoft documentation.

Registering a COM Object

You can register your server object as an in-process or an out-of-process server. For more information on the server types, see In-process, out-of-process, and remote servers.

Note: Before you remove a COM object from your system, you should unregister it.

Registering an in-process server

To register an in-process server (DLL or OCX), choose **Run** ▸ **Register ActiveX Server**.

To unregister an in-process server, choose **Run** ▸ **Unregister ActiveX Server**.

Registering an out-of-process server

To register an out-of-process server, run the server with the **/regserver** command-line option. You can set command-line options with the **Run** ▸ **Parameters** dialog box. You can also register the server by running it.

To unregister an out-of-process server, run the server with the **/unregserver** command-line option.

As an alternative, you can use the tregsvr command from the command line or run the regsvr32 command from the operating system.

Note: If the COM server is intended for use under COM+, you should install it in a COM+ application rather than register it. (Installing the object in a COM+ application automatically takes care of registration.)

Testing and Debugging the Application

Once you have created a COM server application, you will want to test it before you deploy it.

To test and debug your COM server application,

1. Turn on debugging information using the **Compiler** page on the **Project** ▸ **Options** dialog box, if necessary. Also, turn on **Integrated Debugging** in the **Tools** ▸ **Options** ▸ **Debugger Options** dialog.
2. For an in-process server, choose **Run** ▸ **Parameters**, type the name of the Automation controller in the **Host Application** box, and choose OK.
3. Choose **Run** ▸ **Run**.
4. Set breakpoints in the Automation server.
5. Use the Automation controller to interact with the Automation server.

The Automation server pauses when the breakpoints are reached.

Note: As an alternate approach, if you are also writing the Automation controller, you can debug into an in-process server by enabling COM cross-process support. Use the **Borland Debuggers** page of the **Tools** ▸ **Options** ▸ **Debugger Options** dialog to enable cross-process support.

Creating an Active Server Page

Creating Active Server Pages: Overview

If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web-based client/server applications. Active Server Pages let you write a script that gets called every time the server loads the Web page. This script can, in turn, call on Automation objects to obtain information that it includes in a generated HTML page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and use this control to access data that gets updated every time the server loads the Web page.

On the client side, the ASP acts like a standard HTML document and can be viewed by users on any platform using any Web Browser.

ASP applications are analogous to applications you write using Delphi's Web broker technology. For more information about the Web broker technology, see Creating Internet server applications. ASP differs, however, in the way it separates the UI design from the implementation of business rules or complex application logic.

- The UI design is managed by the Active Server Page. This is essentially an HTML document, but it can include embedded script that calls on Active Server objects to supply it with content that reflects your business rules or application logic.
- The application logic is encapsulated by Active Server objects that expose simple methods to the Active Server Page, supplying it with the content it needs.

Note: Although ASP provides the advantage of separating UI design from application logic, its performance is limited in scale. For Web sites that respond to extremely large numbers of clients, an approach based on the Web broker technology is recommended instead.

The script in your Active Server Pages and the Automation objects you embed in an active server page can make use of the ASP intrinsics (built-in objects that provide information about the current application, HTTP messages from the browser, and so on).

The following topics show how to create an Active Server Object using the Delphi Active Server Object wizard. This special Automation control can then be called by an Active Server Page and supply it with content.

Here are the steps for creating an Active Server Object:

- Create an Active Server Object for the application.
- Define the Active Server Object's interface.
- Register the Active Server Object.
- Test and debug the application.

Creating an Active Server Object

An Active Server Object is an Automation object that has access to information about the entire ASP application and the HTTP messages it uses to communicate with browsers. It descends from TASPObject or TASPMTSObject (which is in turn a descendant of TAutoObject), and supports Automation protocols, exposing itself for other applications (or the script in the Active Server page) to use. You create an Active Server Object using the Active Server Object wizard.

Your Active Server Object project can be either an executable (exe) or library (dll), depending on your needs. However, you should be aware of the drawbacks of using an out-of-process server.

To display the Active Server Object wizard:

1. Choose **File** ▸ **New** ▸ **Other**.
2. Select the folder labeled ActiveX under Delphi Projects.
3. Double-click the Active Server Object icon.

   In the wizard, give your new Active Server Object a name, and specify the instancing and threading models you want to support. These details influence the way your object can be called. You must write the implementation so that it adheres to the model (for example, avoiding thread conflicts).

   The thing that makes an Active Server Object unique is its ability to access information about the ASP application and the HTTP messages that pass between the Active Server page and client Web browsers. This information is accessed using the ASP intrinsics. In the wizard, you can specify how your object accesses these by setting the Active Server Type:

- If you are working with IIS 3 or IIS 4, you use Page Level Event Methods. Under this model, your object implements the methods, OnStartPage and OnEndPage, which are called when the Active Server page loads and unloads. When your object is loaded, it automatically obtains an *IScripting*Context interface, which it uses to access the ASP intrinsics. These interfaces are, in turn, surfaced as properties inherited from the base class (*TASPObject*).
- If you are working with IIS5 or later, you use the Object Context type. Under this model, your object fetches an IObjectContext interface, which it uses to access the ASP intrinsics. Again, these interfaces are surfaced as properties in the inherited base class (TASPMTSObject). One advantage of this latter approach is that your object has access to all of the other services available through *IObjectContext*. To access the *IObjectContext* interface, simply call *GetObjectContext* (defined in the mtx unit) as follows: `ObjectContext := GetObjectContext;` For more information about the services available through *IObjectContext*, see Creating MTS or COM+ objects

You can tell the wizard to generate a simple ASP page to host your new Active Server Object. The generated page provides a minimal script (written in VBScript) that creates your Active Server Object based on its ProgID, and indicates where you can call its methods. This script calls **Server.CreateObject** to launch your Active Server Object.

Note: Although the generated test script uses VBScript, Active Server Pages also can be written using Jscript.

When you exit the wizard, a new unit is added to the current project that contains the definition for the Active Server Object. In addition, the wizard adds a type library project and opens the Type Library editor. Now you can expose the properties and methods of the interface through the type library as described in Defining a COM object's interface As you write the implementation of your object's properties and methods, you can take advantage of the ASP intrinsics to obtain information about the ASP application and the HTTP messages it uses to communicate with browsers.

The Active Server Object, like any other Automation object, implements a dual interface, which supports both early (compile-time) binding through the VTable and late (runtime) binding through the IDispatch interface.

Using the ASP Intrinsics

The ASP intrinsics are a set of COM objects supplied by ASP to the objects running in an Active Server Page. They let your Active Server Object access information that reflects the messages passing between your application and the Web browser, as well as a place to store information that is shared among Active Server Objects that belong to the same ASP application.

To make these objects easy to access, the base class for your Active Server Object surfaces them as properties. For a complete understanding of these objects, see the Microsoft documentation. However, the following topics provide a brief overview.

Application

The Application object is accessed through an *IApplicat*ionObject interface. It represents the entire ASP application, which is defined as the set of all .asp files in a virtual directory and its subdirectories. The Application object can be shared by multiple clients, so it includes locking support that you should use to prevent thread conflicts.

IApplicationObject includes the following:

IApplicationObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the application using script commands. This interface has two methods, *Remove* and *RemoveAll*, that you can use to delete one or all objects from the list.
StaticObjects property	Lists all the objects that were added to the application with the <OBJECT> tag.
Lock method	Prevents other clients from locking the Application object until you call Unlock. All clients should call Lock before accessing shared memory (such as the properties).
Unlock method	Releases the lock that was set using the Lock method.
Application_OnEnd event	Occurs when the application quits, after the Session_OnEnd event. The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.
Application_OnStart event	Occurs before the new session is created (before Session_OnStart). The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.

Request

The Request object is accessed through an *IReque*st interface. It provides information about the HTTP request message that caused the Active Server Page to be opened.

IRequest includes the following:

IRequest interface members

Property, Method, or Event	Meaning
ClientCertificate property	Indicates the values of all fields in the client certificate that is sent with the HTTP message.
Cookies property	Indicates the values of all Cookie headers on the HTTP message.
Form property	Indicates the values of form elements in the HTTP body. These can be accessed by name.
QueryString property	Indicates the values of all variables in the query string from the HTTP header.
ServerVariables property	Indicates the values of various environment variables. These variables represent most of the common HTTP header variables.
TotalBytes property	Indicates the number of bytes in the request body. This is an upper limit on the number of bytes returned by the BinaryRead method.

BinaryRead method	Retrieves the content of a Post message. Call the method, specifying the maximum number of bytes to read. The resulting content is returns as a Variant array of bytes. After calling BinaryRead, you can't use the Form property.

Response

The Request object is accessed through an *IRe*sponse interface. It lets you specify information about the HTTP response message that is returned to the client browser.

IResponse includes the following:

IResponse interface members

Property, Method, or Event	Meaning
Cookies property	Determines the values of all Cookie headers on the HTTP message.
Buffer property	Indicates whether page output is buffered When page output is buffered, the server does not send a response to the client until all of the server scripts on the current page are processed.
CacheControl property	Determines whether proxy servers can cache the output in the response.
Charset property	Adds the name of the character set to the content type header.
ContentType property	Specifies the HTTP content type of the response message's body.
Expires property	Specifies how long the response can be cached by a browser before it expires.
ExpiresAbsolute property	Specifies the date and time when the response expires.
IsClientConnected property	Indicates whether the client has disconnected from the server.
Pics property	Set the value for the pics-label field of the response header.
Status property	Indicates the status of the response. This is the value of an HTTP status header.
AddHeader method	Adds an HTTP header with a specified name and value.
AppendToLog method	Adds a string to the end of the Web server log entry for this request.
BinaryWrite method	Writes raw (uninterpreted) information to the body of the response message.
Clear method	Erases any buffered HTML output.
End method	Stops processing the .asp file and returns the current result.
Flush method	Sends any buffered output immediately.
Redirect method	Sends a redirect response message, redirecting the client browser to a different URL.
Write method	Writes a variable to the current HTTP output as a string.

Session

The Session object is accessed through the *ISessionObject* interface. It allows you to store variables that persist for the duration of a client's interaction with the ASP application. That is, these variables are not freed when the client moves from page to page within the ASP application, but only when the client exits the application altogether.

ISessionObject includes the following:

ISessionObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the session using the <OBJECT> tag. You can access any variable in the list by name, or call the Contents object's *Remove* or *RemoveAll* method to delete values.
StaticObjects property	Lists all the objects that were added to the session with the <OBJECT> tag.
CodePage property	Specifies the code page to use for symbol mapping. Different locales may use different code pages.
LCID property	Specifies the locale identifier to use for interpreting string content.
SessionID property	Indicates the session identifier for the current client.
TimeOut property	Specifies the time, in minutes, that the session persists without a request (or refresh) from the client until the application terminates.
Abandon method	Destroys the session and releases its resources.
Session_OnEnd event	Occurs when the session is abandoned or times out. The only intrinsics available are Application, Server, and Session. The event handler must be written in VBScript or JScript.
Session_OnStart event	Occurs when the server creates a new session is created (after Application_OnStart but before running the script on the Active Server Page). All intrinsics are available. The event handler must be written in VBScript or JScript.

Server

The Server object is accessed through an *IServer* interface. It provides various utilities for writing your ASP application.

IServer includes the following:

IServer interface members

Property, Method, or Event	Meaning
ScriptTimeOut property	Same as the TimeOut property on the Session object.
CreateObject method	Instantiates a specified Active Server Object.
Execute method	Executes the script in a specified .asp file.
GetLastError method	Returns an ASPError object that describes the error condition.
HTMLEncode method	Encodes a string for use in an HTML header, replacing reserved characters by the appropriate symbolic constants.
MapPath method	Maps a specified virtual path (an absolute path on the current server or a path relative to the current page) into a physical path.
Transfer method	Sends all of the current state information to another Active Server Page for processing.
URLEncode method	Applies URL encoding rules, including escape characters, to a specified string

Creating ASPs for In-process or Out-of-process Servers

You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process server, depending on your requirements. However, launching in-process servers is more common.

Unlike most in-process servers, an Active Server Object in an in-process server does not run in the client's process space. Instead, it runs in the IIS process space. This means that the client does not need to download your application

(as, for example, it does when you use ActiveX objects). In-process component DLLs are faster and more secure than out-of-process servers, so they are better suited for server-side use.

Because out-of-process servers are less secure, it is common for IIS to be configured to *not* allow out-of-process executables. In this case, creating an out-of-process server for your Active Server Object would result in an error similar to the following:

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

Also, out-of-process components often create individual server processes for each object instance, so they are slower than CGI applications. They do not scale as well as component DLLs.

If performance and scalability are priorities for your site, in-process servers are highly recommended. However, Intranet sites that receive moderate to low traffic may use an out-of-process component without adversely affecting the site's overall performance.

Registering an Active Server Object

You can register the Active Server Page as an in-process or an out-of-process server. However, in-process servers are more common.

Note: When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.

Registering an in-process server

To register an in-process server (DLL or OCX), choose **Run** ▸ **Register ActiveX Server**.

To unregister an in-process server, choose **Run** ▸ **Unregister ActiveX Server**.

Registering an out-of-process server

To register an out-of-process server, run the server with the /regserver command-line option. You can also register the server by running it.

To unregister an out-of-process server, run the server with the /unregserver command-line option.

Testing and Debugging the Active Server Page Application

Debugging any in-process server such as an Active Server Object is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual.

To test and debug an Active Server Object,

1. Turn on debugging information using the Compiler tab on the **Project** ▸ **Options** dialog box, if necessary. Also, turn on Integrated Debugging in the **Tools** ▸ **Options** ▸ **Debugger Options** dialog.
2. Choose **Run** ▸ **Parameters**, type the name of your Web Server in the Host Application box, and choose OK.
3. Choose **Run** ▸ **Run**.
4. Set breakpoints in the Active Server Object implementation.

5 Use the Web browser to interact with the Active Server Page.

The debugger pauses when the breakpoints are reached.

Using ActiveX controls

Elements of an ActiveX Control

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, a corresponding COM object wrapper that exposes properties, methods, and events, and one or more associated type libraries.

VCL control

The underlying implementation of an ActiveX control in Delphi is a VCL control. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

The underlying VCL control must be a descendant of *TWinControl*, because it must have a window that can be parented by the host application. When you create an Active form, this object is a descendant of TActiveForm.

Note: The ActiveX control wizard lists the available *TWinControl* descendants from which you can choose to make an ActiveX control. This list does not include all *TWinControl* descendants, however. Some controls, such as *THeaderControl*, are registered as incompatible with ActiveX (using the RegisterNonActiveXprocedure procedure) and do not appear in the list.

ActiveX wrapper

The actual COM object is an ActiveX wrapper object for the VCL control. For Active forms, this class is always TActiveFormControl. For other ActiveX controls, it has a name of the form *TVCLClass*X, where *TVCLClass* is the name of the VCL control class. Thus, for example, the ActiveX wrapper for *TButton* would be named *TButtonX*.

The wrapper class is a descendant of TActiveXControl, which provides support for the ActiveX interfaces. The ActiveX wrapper inherits this support, which allows it to forward Windows messages to the VCL control and parent its window in the host application.

The ActiveX wrapper exposes the VCL control's properties and methods to clients via its default interface. You must implement the wrapper class' properties and methods, delegating method calls to the underlying VCL control. You must also provide the wrapper class with methods that fire the VCL control's events on clients and assign these methods as event handlers on the VCL control.

Type library

You must generate a type library for your ActiveX control that contains the type definitions for the wrapper class, its default interface, and any type definitions that these require. This type information provides a way for your control

to advertise its services to host applications. You can view and edit this information using the Type Library editor. Although this information is stored in a separate, binary type library file (.TLB extension), you may also compile it into the ActiveX control DLL as a resource.

Property page

You can optionally give your ActiveX control a property page. The property page allows the user of a host (client) application to view and edit your control's properties. You can group several properties on a page, or use a page to provide a dialog-like interface for a property. For information on how to create property pages, see Creating a property page for an ActiveX control.

Designing an ActiveX Control

When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating custom controls, see Creating custom components.

When designing the VCL control, keep in mind that it will be embedded in another application; this control is not an application in itself. For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application.

In addition, you should make sure that the types for all properties and methods you want your object to expose to clients are Automation-compatible, because the ActiveX control's interface must support *IDispatch*. The wizards do not add any methods to the wrapper class's interface that have parameters that are not Automation-compatible.

The wizards implement all the necessary ActiveX interfaces required using the COM wrapper class. They also surface all Automation-compatible properties, methods, and events through the wrapper class's default interface. Once a wizard has generated the COM wrapper class and its interface, you can use the Type Library editor to modify the default interface or augment the wrapper class by implementing additional interfaces.

Generating an ActiveX Control Based On a VCL Form

Unlike other ActiveX controls, Active Forms are not first designed and then wrapped by an ActiveX wrapper class. Instead, the ActiveForm wizard generates a blank form that you design later when the wizard leaves you in the Form Designer.

When an ActiveForm is deployed on the Web, Delphi creates an HTML page to contain the reference to the ActiveForm and specify its location on the page. The ActiveForm can then displayed and run from a Web browser. Inside the browser, the form behaves just like a stand-alone Delphi form. The form can contain any VCL components or ActiveX controls, including custom-built VCL controls.

To start the ActiveForm wizard,

1. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
2. Select the tab labeled ActiveX.
3. Double-click the ActiveForm icon.

On the Active Form wizard, you can't specify the name of the VCL class to wrap. This is because Active forms are always based on TActiveForm.

You can change the default names for the CoClass, implementation unit, and ActiveX library project. Similarly, this wizard lets you indicate whether you want your Active Form to require a license, whether it should include version information, and whether you want an About box form.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, see Working with type libraries.
- A form that descends from TActiveForm. This form appears in the form designer, where you can use it to visually design the Active Form that appears to clients. Its implementation appears in the generated implementation unit. In the initialization section of the implementation unit, a class factory is created, setting up TActiveFormControl as the ActiveX wrapper for this form.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

At this point, you can add controls and design the form as you like.

After you have designed and compiled the ActiveForm project into an ActiveX library (which has the OCX extension), you can deploy the project to your Web server and Delphi creates a test HTML page with a reference to the ActiveForm.

Licensing ActiveX Controls

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically for controls created at runtime.

To provide design-time licenses, a key is created for the control , which is stored in a file with the same name as the project with the LIC extension. This .LIC file is added to the project. The user of the control must have a copy of the .LIC file to open the control in a development environment. Each control in the project that has Make Control Licensed checked has a separate key entry in the .LIC file.

To support runtime licenses, the wrapper class implements two methods, *GetLicenseString* and *GetLicenseFilename*. These return the license string for the control and the name of the .LIC file, respectively. When a host application tries to create the ActiveX control, the class factory for the control calls these methods and compares the string returned by *GetLicenseString* with the string stored in the .LIC file.

Runtime licenses for the Internet Explorer require an extra level of indirection because users can view HTML source code for any Web page, and because an ActiveX control is copied to the user's computer before it is displayed. To create runtime licenses for controls used in Internet Explorer, you must first generate a license package file (LPK file) and embed this file in the HTML page that contains the control. The LPK file is essentially an array of ActiveX control CLSIDs and license keys.

Note: To generate the LPK file, use the utility, LPK_TOOL.EXE, which you can download from the Microsoft Web site (www.microsoft.com).

To embed the LPK file in a Web page, use the HTML objects, <OBJECT> and <PARAM> as follows:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
<PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

The CLSID identifies the object as a license package and PARAM specifies the relative location of the license package file with respect to the HTML page.

When Internet Explorer tries to display the Web page containing the control, it parses the LPK file, extracts the license key, and if the license key matches the control's license (returned by *GetLicenseString*), it renders the control on the page. If more than one LPK is included in a Web page, Internet Explorer ignores all but the first.

For more information, look for Licensing ActiveX Controls on the Microsoft Web site.

Customizing the ActiveX Control's Interface

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library. You can use the Type Library editor as described in Using the Type Library Editor. Remember that when you add events, they should be added to the Events interface, not the ActiveX control's default interface.

Note: You can add unpublished properties to your ActiveX control's interface. Such properties can be set at runtime and will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property at design time, the changes are not reflected when the control is run. If the source is a VCL object and the property is not already published, you can make properties persistent by creating a descendant of the VCL object and publishing the property in the descendant.

You may also choose not to expose all of the VCL control's properties, methods, and events to host applications. You can use the Type Library editor to remove these from the interfaces that the wizard generated. When you remove properties and methods from an interface using the Type Library editor, the Type Library editor does not remove them from the corresponding implementation class. Edit the ActiveX wrapper class in the implementation unit to remove these after you have changed the interface in the Type Library editor.

Warning: Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

Tip: It is a good idea to check the methods that the wizard adds to your ActiveX wrapper class. Not only does this give you a chance to note where the wizard omitted any data-aware properties or methods that were not Automation-compatible, it also lets you detect methods for which the wizard could not generate an implementation. Such methods appear with a comment in the implementation that indicates the problem.

Adding Additional Properties, Methods, and Events

You can add additional properties, methods, and events to the control using the type library editor. The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies depends on whether you have added a property or method or whether you have added an event.

How Delphi Adds Properties

The ActiveX wrapper class implements properties in its interface using read and write access methods. That is, the wrapper class has COM properties, which appear on an interface as getter and/or setter methods. Unlike VCL properties, you do not see a "property" declaration on the interface for COM properties. Rather, you see methods that are flagged as property access methods. When you add a property to the ActiveX control's default interface, the wrapper class definition (which appears in the _TLB unit that is updated by the Type Library editor) gains one or two new methods (a getter and/or setter) that you must implement, just as when you add a method to the interface, the wrapper class gains a corresponding method for you to implement. Thus, adding properties to the wrapper class's interface is essentially the same as adding methods: the wrapper class definition gains new skeletal method implementations for you to complete.

Note: For details on what appears in the generated _TLB unit, see Code generated when you import type library information.

For example, consider a *Caption* property, of type TCaption in the underlying VCL object. To Add this property to the object's interface, you enter the following when you add a property to the interface via the type library editor:

```
[Delphi]
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi adds the following declarations to the wrapper class:

```
[Delphi]
function Get_Caption: WideString; safecall;
procedure Set_Caption(const Value: WideString); safecall;
```

```
[C++]
STDMETHOD(get_Caption(BSTR* Value));
STDMETHOD(set_Caption(BSTR Value));
```

In addition, it adds skeletal method implementations for you to complete:

```
[Delphi]
function TButtonX.Get_Caption: WideString;
begin
end;
procedure TButtonX.Set_Caption(Value: WideString);
begin
end;
```

```
[C++]
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

Typically, you can implement these methods by simply delegating to the associated VCL control, which can be accessed using the *FDelphiControl* member of the wrapper class:

```
[Delphi]
function TButtonX.Get_Caption: WideString;
begin
   Result := WideString(FDelphiControl.Caption);
end;
```

```
procedure TButtonX.Set_Caption(const Value: WideString);
begin
   FDelphiControl.Caption := TCaption(Value);
end;
```

```
[C++]
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
try
{
*Value = WideString(m_VclCtl->Caption).Copy();
}
catch(Exception &e)
{
return Error(e.Message.c_str(), IID_IButtonX);
}
return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
    m_VclCtl->Caption = AnsiString(Value);
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

In some cases, you may need to add code to convert the COM data types to native Delphi types. The preceding example manages this with typecasting.

Note: Because the Automation interface methods are declared **safecall**, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of safecall methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.

How Delphi Adds Events

The ActiveX control can fire events to its container in the same way that an automation object fires events to clients. This mechanism is described in Managing events in your Automation object.

If the VCL control you are using as the basis of your ActiveX control has any published events, the wizards automatically add the necessary support for managing a list of client event sinks to your ActiveX wrapper class and define the outgoing dispinterface that clients must implement to respond to events.

You add events to this outgoing dispinterface. To add an event in the type library editor, select the event interface and click on the method icon. Then manually add the list of parameters you want include using the parameter page.

Next, you must declare a method in your wrapper class that is of the same type as the event handler for the event in the underlying VCL control. This is not generated automatically, because Delphi does not know which event handler you are using:

```
[Delphi]
procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

Implement this method to use the host application's event sink, which is stored in the wrapper class's *FEvents* member:

```
[Delphi]
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
    TempKey := Smallint(Key); {cast to an OleAutomation compatible type }
    if FEvents <> nil then
      FEvents.OnKeyPress(TempKey)
    Key := Char(TempKey);
  end;
```

```
[C++]
void __fastcall TButtonXImpl::KeyPressEvent(TObject *Sender, char &Key)
{
short TempKey;
TempKey = (short)Key;
Fire_OnKeyPress(&TempKey);
Key = (short)TempKey;
};
```

Note: When firing events in an ActiveX control, you do not need to iterate through a list of event sinks because the control only has a single host application. This is simpler than the process for most Automation servers.

Finally, you must assign this event handler to the underlying VCL control, so that it is called when the event occurs. You make this assignment in the *InitializeControl* method:

```
[Delphi]
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
```

```
[C++]
void InitializeControl()
{
  m_VclCtl->OnClick = ClickEvent;
  m_VclCtl->OnKeyPress = KeyPressEvent;
}
```

Enabling Simple Data Binding with the Type Library

With simple data binding, you can bind a property of your ActiveX control to a field in a database. To do this, the ActiveX control must communicate with its host application about what value represents field data and when it changes. You enable this communication by setting the property's binding flags using the Type Library editor.

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies its container (the client host application) that the value has changed and requests that the database record be updated. The container interacts with the database and then notifies the control whether it succeeded or failed to update the record.

Note: The container application that hosts your ActiveX control is responsible for connecting the data-aware properties you enable in the type library to the database.

Use the type library to enable simple data binding,

1. On the toolbar, click the property that you want to bind.
2. Choose the flags page.
3. Select the following binding attributes:

Binding attribute	Description
Bindable	Indicates that the property supports data binding. If marked bindable, the property notifies its container when the property value has changed.
Request Edit	Indicates that the property supports the OnRequestEdit notification. This allows the control to ask the container if its value can be edited by the user.
Display Bindable	Indicates that the container can show users that this property is bindable.
Default Bindable	Indicates the single, bindable property that best represents the object. Properties that have the default bind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Immediate Bindable	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

4. Click the Refresh button on the toolbar to update the type library.

   To test a data-binding control, you must register it first.

   For example, to convert a *TEdit* control into a data-bound ActiveX control, create the ActiveX control from a *TEdit* and then change the Text property flags to Bindable, Display Bindable, Default Bindable, and Immediate Bindable.

   After the control is registered and imported, it can be used to display data.

Creating a Property Page for an ActiveX Control

A property page is a dialog box similar to the Delphi **Object Inspector** in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties.

Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.

The process of creating a property page is similar to creating a form, you

1. Create a new property page.
2. Add controls to the property page.
3. Associate the controls the property page with the properties of an ActiveX control.

4 Connect the property page to the ActiveX control.

Note: When adding properties to an ActiveX control or ActiveForm, you must publish the properties that you want to persist. If they are not published in the underlying VCL control, you must make a custom descendant of the VCL control that redeclares the properties as published and then create an ActiveX control from the descendant class.

Creating a New Property Page

You use the Property Page wizard to create a new property page.

To create a new property page,

1 Choose **File** ▸ **New** ▸ **Other**.

2 Select the ActiveX folder under Delphi Projects .

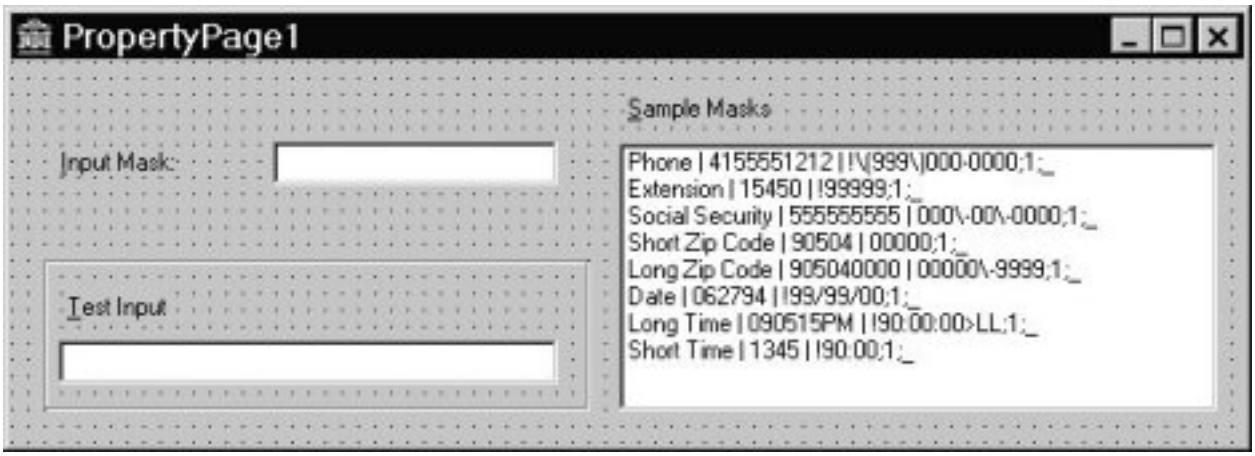
3 Double-click the Property Page icon in the right pane.

The wizard creates a new form and implementation unit for the property page. The form is a descendant of TPropertyPage, which lets you associate the form with the ActiveX control whose properties it edits.

Adding Controls to a Property Page

You must add a control to the property page for each property of the ActiveX control that you want the user to access.

For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.



The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.

Associating Property Page Controls with ActiveX Control Properties

After adding the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's UpdatePropertyPage and UpdateObject methods.

Updating the Property Page

Add code to the *UpdatePropertyPage* method to update the control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties.

You can access the ActiveX control using the property page's OleObject property, which is an *OleVariant* that contains the ActiveX control's interface.

For example, the following code updates the property page's edit control (InputMask) with the current value of the ActiveX control's *EditMask* property:

```
[Delphi]
procedure TPropertyPage1.UpdatePropertyPage;
begin
   { Update your controls from OleObject }
   InputMask.Text := OleObject.EditMask;
end;
```

For example, the following code updates the property page's edit control (InputMask) with tthe current value of the ActiveX control's EditMask property:

```
[C++]
void __fastcall TPropertyPage1::UpdatePropertyPage(void)
{
InputMask->Text = OleObject.OlePropertyGet("EditMask");
}
```

Note: It is also possible to write a property page that represents more than one ActiveX control. In this case, you don't use the *OleObject* property. Instead, you must iterate through a list of interfaces that is maintained by the *OleObjects* property.

Updating the Object

Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values.

You use the OleObject property to access the ActiveX control.

For example, the following code sets the *EditMask* property of the ActiveX control using the value in the property page's edit box control (InputMask):

```
[Delphi]
procedure TPropertyPage1.UpdateObject;
begin
   {Update OleObject from your control }
   OleObject.EditMask := InputMask.Text;
end;
```

```
[C++]
void __fastcall TPropertyPage1::UpdateObject(void)
{
  // Update OleObject from your control
  OleObject.OlePropertySet<WideString>("EditMask", WideString(InputMast->Text).Copy());
}
```

Connecting a Property Page to an ActiveX Control

To connect a property page to an ActiveX control,

1 Add DefinePropertyPage with the GUID constant of the property page as the parameter to the DefinePropertyPages method implementation in the control's implementation for the unit. For example,

```
[Delphi]
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
   DefinePropertyPage(Class_PropertyPage1);
end;
```

```
[C++]
BEGIN_PROPERTY_MAP(TActiveFormXImpl)
// Define property pages here.  Property pages are defined using
// the PROP_PAGE macro with the class id of the page.  For example,
// PROP_PAGE(CLSID_ActiveFormXPage)
PROP_PAGE(CLSID_PropertyPage1)
END_PROPERTY_MAP()
```

The GUID constant, Class_PropertyPage1, of the property page can be found in the property pages unit.

The GUID is defined in the property page's implementation unit .

2 Add the property page unit to the **uses** clause of the controls implementation unit.

Component Writer's Guide

Introduction to component creation

Overview of Component Creation

This set of topics provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components.

The main topics discussed are

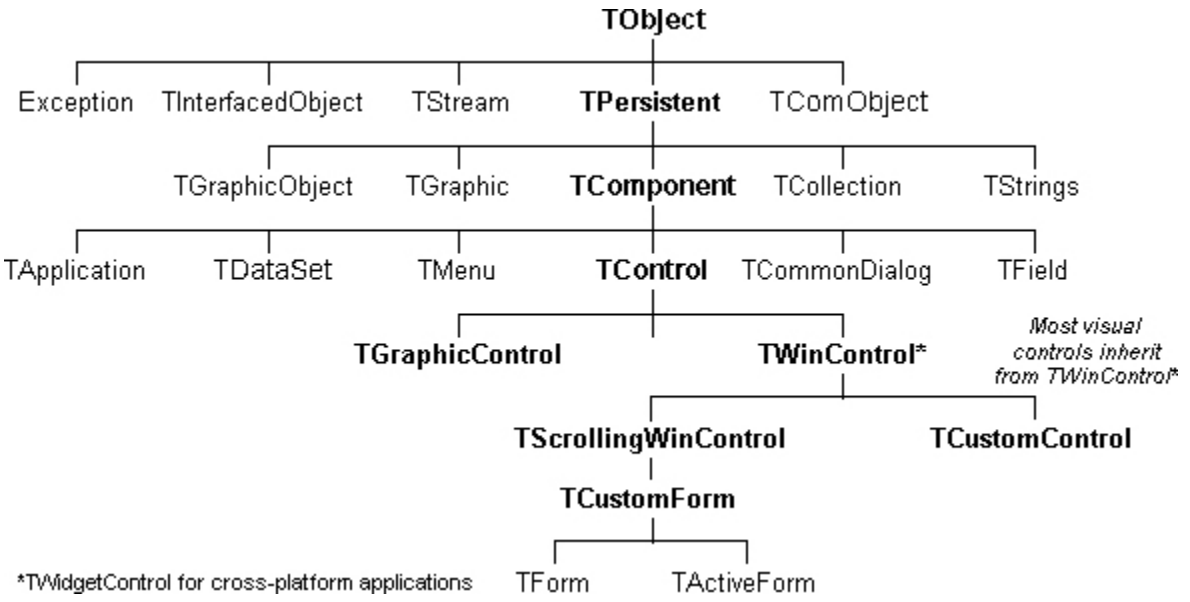
- Class library
- Components and classes
- Creating components
- What goes into a component?
- Creating a new component
- Testing uninstalled components
- Testing installed components

For information on installing new components, see Installing component packages.

Class library

Delphi's components reside in the Visual Component Library (VCL) . The following figure shows the relationship of selected classes that make up the VCL hierarchy. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Object-oriented programming for component writers

The *TComponent* class is the shared ancestor of every component in the component library. *TComponent* provides the minimal properties and events necessary for a component to work in the IDE. The various branches of the library provide other, more specialized capabilities.



When you create a component, you add to the component library by deriving a new class from one of the existing class types in the hierarchy.

Components and Classes

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

Creating Components

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component in several ways:

- Modifying existing controls
- Creating windowed controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

The following table summarizes the different kinds of components and the classes you use as starting points for each.

Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*

Create a windowed control	*TWinControl*
Create a graphic control	*TGraphicControl*
Subclassing a control	Any Windows control
Create a nonvisual component	*TComponent*

You can also derive classes that are not components and cannot be manipulated on a form, such as *TRegIniFile* and *TFont*.

Modifying Existing Controls

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided in the component library.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the component library includes an abstract class (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWinControl*class and reinvent all the list box functions, the component library provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

The section Creating properties explains publishing inherited properties. The section Modifying an existing component and the section Customizing a grid show examples of modifying existing controls.

Creating Original Controls

Windowed controls in the component library are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets the operating system identify and operate on the control. If using VCL controls, the handle allows the control to receive input focus and can be passed to Windows API functions. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.-->

All windowed controls descend from the *TWinControl*class. These include most standard windowed controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWinControl*, Delphi provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized windowed control that makes it easier to draw complex visual images.

The section Customizing a grid presents an example of creating a windowed control.

Creating Graphic Controls

If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse messages.

You can create custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on and on Windows, handles *WM_PAINT* messages; all you need to do is override the *Paint* method.

The section Creating a graphic control presents an example of creating a graphic control.

Subclassing Windows Controls

In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

You can create a component "wrapper" around any existing window class. So if you already have a library of custom controls that you want to use in Delphi applications, you can create Delphi components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing Windows controls, see the components in the StdCtls unit that represent standard Windows controls, such as *TEdit*.

Creating Nonvisual Components

Nonvisual components are used as interfaces for elements like databases (*TDataSet or TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.

What Goes into a Component?

To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies
- Setting properties, methods, and events
- Encapsulating graphics
- Registering components

Removing Dependencies

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An example of removing dependencies is the *Handle* property of *TWinControl*. If you have written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a windowed control until you have created it by calling the *CreateWindow* API function. Delphi windowed controls relieve users from this concern by ensuring that a valid window handle is always available when needed. By using a property to represent the window handle, the control can check whether the window has been created; if the handle is not valid, the control creates a window and returns the handle. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the window, Delphi components allow developers to focus on what they really want to do. Before passing a window handle to an API function, you do not need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

Setting Properties, Methods, and Events

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a section devoted to it in this file, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

The section Overview of component creation explains how to add properties to your components.

Methods

Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's constructor method (*Create*) is a class method. Component methods are procedures and functions that operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

The section Creating methods explains how to add methods to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

The section Creating events explains how to implement standard events and how to define new ones.

Encapsulating Graphics

Delphi simplifies Windows graphics by encapsulating various graphics tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is like a Windows device context, but it takes care of all the bookkeeping for you.

If you have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them.

With Delphi, you do not have to worry about these things. To draw on a form or other component, you access the component's *Canvas* property. If you want to customize a pen or brush, you set its color or style. When you finish, Delphi disposes of the resources. Delphi caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You still have full access to the Windows GDI, but you will often find that your code is simpler and runs faster if you use the canvas built into Delphi components.

How graphics images work in the component depends on the canvas of the object from which your component descends. Graphics features are detailed in the section Using graphics in components.

Registering Components

Before you can install your components in the IDE, you have to register them. Registration tells Delphi where to place the component on the Tool palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Registering components.

Creating a New Component

This topic describes how to create and setup a component.

To create a component, follow these steps:

1. Creating a unit file
2. Deriving the component
3. Registering the component

Now you will have a minimally functional component ready to install on the Tool palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Tool palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1. Create a unit for the new component.
2. Derive your component from an existing component type.
3. Add properties, methods, and events.
4. Register your component with the IDE.
5. Create a bitmap for the component.
6. Create a package (a special dynamic-link library) so that you can install your component in the IDE.
7. Create a Help file for your component and its properties, methods, and events.

Note: Creating a Help file to instruct component users on how to use the component is optional.

When you finish, the complete component includes the following files:

- A package (.BPL) or package collection (.DPC) file
- A compiled package (.DCP) file
- A compiled unit (.DCU) file
- A palette bitmap (.DCR) file
- A Help (.HLP) file

You can also create a bitmap to represent your new component. .

Creating a Component with the Component Wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify:

- The class from which the component is derived.
- The class name for the new component.
- The Tool palette category where you want it to appear.
- The name of the unit in which the component is created.
- The search path where the unit is found.
- The name of the package in which you want to place the component.

The Component wizard performs the same tasks you would when creating a component manually:

- Creating a unit.
- Deriving the component.
- Registering the component.

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To add a new component with the Component Wizard

1. To start the Component wizard, choose one of these two methods:

- Choose **Component** ▸ **New VCL Component**.

- Choose **File** ▸ **New** ▸ **Other**, goto the **Delphi Projects** ▸ **Delphi Files** page and double-click Component.

2 Fill in the fields in the Component wizard:

- In the Ancestor Type field, specify the class from which you are deriving your new component.
- In the Class Name field, specify the name of your new component class.
- In the Palette Page field, specify the category on the Tool palette on which you want the new component to be installed.
- In the Unit file name field, specify the name of the unit you want the component class declared in. If the unit is not on the search path, edit the search path in the Search Path field as necessary.

3 After you fill in the fields in the Component wizard,

Click Install. To place the component in a new or existing package, click **Component** ▸ **Install** and use the dialog box that appears to specify a package.

4 Click OK. The IDE creates a new unit.

Warning: If you derive a component from a class whose name begins with "custom" (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Delphi cannot create instance objects of a class that has abstract properties or methods.

To see the source code for your unit, click **View** ▸ **Units...** (If the Component wizard is already closed, open the unit file in the Code editor by selecting **File** ▸ **Open**.) Delphi creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units.

The unit looks like this:

```
[Delphi]
unit MyControl;
    interface
    uses
      Windows, Messages, SysUtils, Types, Classes, Controls;
    type
      TMyControl = class(TCustomControl)
      private
      { Private declarations }
      protected
      { Protected declarations }
      public
      { Public declarations }
      published
      { Published declarations }
    end;
    procedure Register;
    implementation
    procedure Register;
    begin
      RegisterComponents('Samples', [TMyControl]); //In CLX, use a different page than
'Samples'
    end;
    end.
```

```
[C++]
```

```
//header file
#ifndef NewComponentH
#define NewComponentH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TNewComponent : public TComponent
{
private:
protected:
public:
__fastcall TNewComponent(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
```

```
[C++]

//implementation file
#include <vcl.h>
#pragma hdrstop
#include "NewComponent.h"
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TNewComponent *)
{
new TNewComponent(NULL);
}
//---------------------------------------------------------------------------
__fastcall TNewComponent::TNewComponent(TComponent* Owner)
: TComponent(Owner)
{
}
//---------------------------------------------------------------------------
namespace Newcomponent
{
void __fastcall PACKAGE Register()
{
TComponentClass classes[1] = {__classid(TNewComponent)};
RegisterComponents("Samples", classes, 0); //In CLX use a different page than Samples
}
}
```

Creating a Unit File

A unit is a separately compiled module of Delphi code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a new unit for a component:

1 Choose either:

- **File** ▸ **New** ▸ **Unit**.
- **File** ▸ **New** ▸ **Other** to display the New Items dialog box, select **Delphi Projects** ▸ **Delphi Files** ▸ **Unit**, and choose OK.

The IDE creates a new unit file and opens it in the Code editor.

2 Save the file with a meaningful name.

3 Derive the component class.

To open an existing unit:

1 Choose **File** ▸ **Open** and select the source code unit to which you want to add your component.

> **Note:** When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Tool palette.

2 Derive the component class.

Deriving the Component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. The section Creating components describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in The section Defining new classes.

To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component.

A simple component class is a nonvisual component descended directly from *TComponent*.

Declaring A New Constructor (C++)

Each new component must have a constructor that overrides the constructor of the class from which it was derived. When you write the constructor for your new component, it must always call the inherited constructor.

Within the class declaration, declare a virtual constructor in the public section of the class.

For example,

```
class PACKAGE TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

In the .CPP file, implement the constructor:

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
```

Within the constructor, you add the code you want to execute when the component is created.

Registering the Component

Registration is a simple process that tells the IDE which components to add to its component library, and on which pages of the Tool palette they should appear. For a more detailed discussion of the registration process, see Making components available at design time

To register a component:

1 Add a procedure named *Register* to the **interface** part of the component's unit. *Register* takes no parameters, so the declaration is very simple:

```
[Delphi]
procedure Register;
```

```
[C++]
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
```

```
[C++]
TComponentClass classes[1] = {__classid(TNewComponent)};
```

If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

> **Note:** Although Delphi is a case insensitive language, the *Register* procedure is case sensitive and must be spelled with an uppercase R.

2 Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a Tool palette category and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

Making Source Files Available

Component writers should make all source files used by a component should be located in the same directory. These files include source code files (.pas) and additional project files (.dfm/.xfm, .res, .rc, and .dcr).

The process of adding a component results in the creation of a number of files. These files are automatically put in directories specified in the IDE environment options (use the menu command **Tools** ▸ **Options**, navigate to the **Environment Options** ▸ **Delphi Options** ▸ **Library** page). The .lib files are placed in the DCP output directory. If

adding the component entails creating a new package (as opposed to installing it into an existing package), the .bpl file is put in the BPL output directory.

Testing Uninstalled Components

You can test the runtime behavior of a component before you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing already installed components, see Testing installed components.

You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

To test an uninstalled component,

1. Add the name of component's unit to the form unit's **uses** clause.
2. Add an object field to the form to represent the component.

   This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages.

   Never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.
3. Attach a handler to the form's *OnCreate* event.
4. Construct the component in the form's *OnCreate* handler.

   When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.
5. Assign the *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that contains the control visually; usually it is the form on which the control appears, but it might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the control.

   Warning: If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.
6. Set any other component properties as desired.

Testing Installed Components

You can test the design-time behavior of a component after you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing components that have not yet been installed, see Testing uninstalled components.

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of the IDE:

1. Choose **Project** ▸ **Options** and on the Directories/Conditionals page, set the Debug Source Path to the component's source file.
2. Then select **Tools** ▸ **Options**. On the **Debugger Options** ▸ **Borland Debuggers** ▸ **Language Exceptions** page, enable the exceptions you want to track.
3. Open the component source file and set breakpoints.
4. Select **Run** ▸ **Parameters** and set the Host Application field to the name and location of the Delphi executable file.
5. In the Run Parameters dialog, click the Load button to start a second instance of Delphi.
6. Then drop the components to be tested on the form, which should break on your breakpoints in the source.

Object-oriented programming for component writers

Object-oriented Programming for Component Writers: Overview

If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in the following topics, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

Defining New Classes

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

Deriving New Classes

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

Changing Class Defaults to Avoid Repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Tool palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Modifying an existing component shows an example of changing a component's default properties.

Note: If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

Adding New Capabilities to a Class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Customizing a grid shows an example of customizing an abstract component class.

Declaring a New Component Class

In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. The Creating components topic shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file.

Ancestors, Descendants, and Class Hierarchies

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Delphi derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling Access

There are five levels of *access control* - also called *visibility* - on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

The table below shows the levels of visibility, from most restrictive to most accessible:

Levels of visibility within an object

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details.
protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
automated	Accessible to all code. Automation type information is generated.	OLE automation only.
published	Accessible to all code and accessible from the Object Inspector. Saved in a form file.	Defining the design-time interface.

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

Hiding Implementation Details

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

Defining the Component Writer's Interface

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Note: A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component's protected methods (unless the component is declared in the same unit as the form).

Defining the Runtime Interface

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Defining the Design-time Interface

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Read-only properties cannot be part of the design-time interface because the application developer cannot assign values to them directly. Read-only properties should therefore be public, rather than published.

Dispatching Methods

Dispatch refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Regular Methods (C++)

Class methods are regular (or nonvirtual) unless you specifically declare them as virtual, or unless they override a virtual method in a base class. The compiler can determine the exact address of a regular class member at compile time. This is known as compile-time binding.

A base class regular method is inherited by derived classes. In the following example, an object of type `Derived` can call the method `Regular()` as it were its own method. Declaring a method in a derived class with the same name and parameters as a regular method in the class's ancestor replaces the ancestor's method. In the following

example, when `d->AnotherRegular()` is called, it is being dispatched to the `Derived` class replacement for `AnotherRegular()`.

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};
class Derived : public Base
{
public:
    void AnotherRegular();          // replaces Base::AnotherRegular()
    void Virtual();                 // overrides Base::Virtual()
};
void FunctionOne()
{
  Derived *d;
  d = new Derived;
  d->Regular();                   // Calling Regular() as it were a member of Derived
                                  // The same as calling d->Base::Regular()
  d->AnotherRegular();            // Calling the redefined AnotherRegular(), ...
                                  // ... the replacement for Base::AnotherRegular()
  delete d;
}
void FunctionTwo(Base *b)
{
  b->Virtual();
  b->AnotherRegular();
}
```

Static Methods

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

Virtual Methods

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type.

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

Overriding Methods

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

To override a method in a descendant class, add the directive **override** to the end of the method declaration.

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

Dynamic Methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

Abstract Class Members

When a method is declared as **abstract** in an ancestor class, you should surface it (by redeclaring and implementing it) in any descendant component before you use the new component in applications. On the Win32 platform, Delphi can create instances of a class that contains abstract members. This is not recommended, however, and it is not allowed on the .NET platform. For more information about surfacing inherited parts of classes, see Creating properties and Creating methods.

Classes and Pointers

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

Creating properties

Creating Properties: Overview

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form Designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

Why Create Properties?

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, the Object Inspector supplies the values. The Object Inspector also validates property assignments as soon as they are made.
- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

Types of Properties

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

How properties appear in the Object Inspector

Property type	treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (true if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from *TPersistent*.
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of *TComponent*). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

Publishing Inherited Properties

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

Defining Properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include:

- Property declarations
- Internal data storage
- Direct access
- Access methods
- Default property values

Property Declarations

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.

Internal Data Storage

There are no restrictions on how you store the data for a property. In general, however, Delphi components follow these conventions:

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

Direct Access

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

Access Methods (properties)

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

The Read Method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property.

For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see Creating array properties), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Delphi Language Guide*.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

The Write Method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the *Delphi Language Guide*.)

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a field called *FCount*.

```
[Delphi]
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

```
[C++]
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount )
  {
    FCount = Value;
    Update();
  }
```

Default Property Values

When you declare a property, you can specify a *default value* for it. The VCL uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, the VCL always stores the property.

To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,

```
[Delphi]
property Cool Boolean read GetCool write SetCool default True;
```

```
[C++]
__property bool IsTrue = {read=GetIsTrue, write=SetIsTrue, default=true};
```

Note: Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

Specifying No Default Value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

```
[Delphi]
property FavoriteFlavor string nodefault;
```

```
[C++]
__property int NewInteger = {nodefault};
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

Creating Array Properties

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

Creating Properties for Subcomponents

By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property.

However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

Note: When a component that has subcomponents is streamed, the subcomponents will have their csLoading flag set and their Loaded method called. This can create a complication for any subcomponent properties that are writable. If you allow your subcomponent property to be assigned to an external component reference, then you cannot free your subcomponent until it's owner's Loaded method is called. Otherwise, the streaming system will attempt to call the subcomponent's Loaded method after the subcomponent has been freed.

Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil. Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time.

Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method.

Creating Properties for Interfaces

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In Creating properties for subcomponents, the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a result, you can't call its *FreeNotification* method.

To handle this situation, you can call your component's *ReferenceInterface* method:

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component's *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
```

```
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

Storing and Loading Properties

Delphi stores forms and their components in form (.dfm in VCL applications) files. A form file stores the properties of a form and its components. When Delphi developers add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

Using the Store-and-load Mechanism

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying Default Values

Delphi components save their property values only if those values differ from the defaults. If you do not specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when re-declaring a property. In fact, one reason to re-declare a property is to designate a different default value.

```
__property Alignment = {default=taCenter};
```

Note: Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value- that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, assign a value in the constructor method.

Determining What to Store

You can control whether Delphi stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean function.

Initializing After Loading

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note: The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

The following code comes from the *TDatabase* component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;                                    { call the inherited method first}
  try
    if FStreamedConnected then Open                          { reestablish connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then                              { at design time... }
      Application.HandleException(Self)                { let Delphi handle the exception }
    else raise;                                                 { otherwise, reraise }
  end;
end;
```

Storing and Loading Unpublished Properties

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Delphi how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1. Create methods to store and load the property value.
2. Override the DefineProperties method, passing those methods to a filer object.

Creating Methods to Store and Load Property Values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Delphi knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

```
[Delphi]
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

```
[C++]
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
  if (Reader->ReadBoolean())
    MyCompProperty = Reader->ReadComponent(NULL);
}
void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
  if (MyCompProperty)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(MyCompProperty);
  }
  else
    Writer->WriteBoolean(false);
}
```

Overriding the DefineProperties Method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```
[Delphi]
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else   if (MyCompProperty = nil) or
         (TMy5Comp(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name) then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

```
[C++]

void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
{
  // before we do anything, let the base class define its properties.
  // Note that this example assumes that TSampleComponent derives directly from TComponent
  TComponent::DefineProperties(Filer);
  bool WriteValue;
  if (Filer->Ancestor) // check for inherited value
  {
    if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
      WriteValue = (MyCompProperty != NULL);
    else if ((MyCompProperty == NULL) ||
             (((TSampleComponent *)Filer->Ancestor)->MyCompProperty->Name !=
                                                    MyCompProperty->Name))
      WriteValue = true;
    else WriteValue = false;
  }
  else // no inherited value, write property if not null
    WriteValue = (MyCompProperty != NULL);
  Filer->DefineProperty("MyCompProperty ",LoadCompProperty,StoreCompProperty, WriteValue);
end;
```

Creating events

Creating Events: Overview

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

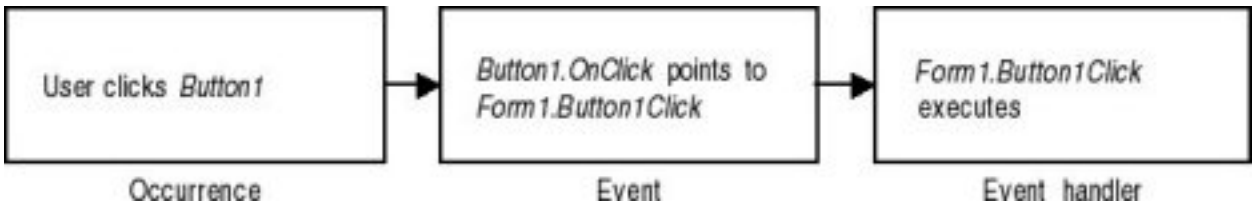
Events are implemented as properties, so you should already be familiar with the material in Creating properties before you attempt to create or change a component's events.

What Are Events?

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer's perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, when you assign a value to the *OnClick* event, the Form Designer generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:



- Events are method pointers.
- Events are properties.
- Event types are method-pointer types.

- Event-handler types are procedures.
- Event handlers are optional.

Events Are closures (C++)

Closures are used to implement events. A closure is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the closure as a place holder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event.

Closures maintain a hidden pointer to a class instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

Events Are Method Pointers

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to a class instance. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

Calling the Click-event Handler

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
[Delphi]
procedure Click; dynamic;
```

```
[C++]
virtual void __fastcall Click(void);
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events Are Properties

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
[Delphi]
type
  TControl = class(TComponent)
  private
```

```
    FOnClick: TNotifyEvent;                 { declare a field to hold the method pointer }
    .
    .
    .
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

```
[C++]
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  .
  .
  .
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  .
  .
  .
};
```

To learn about *TNotifyEvent* and other event types, see the next section, Event types are method-pointer types.

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event Types Are Method-pointer Types

Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that is appropriate, or define one of your own.

Event Handler Types Are Procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

Event Handlers Have A Return Type of void (C++)

Event handlers must have a return type of void. Even though the handler can return only void, you can still get information back from the user's code by passing arguments by reference. When you do this, make sure you assign a valid value to the argument before calling the handler so you do not require the user's code to change the value.

An example of passing arguments by reference to an event handler is the key-pressed event, of type `TKeyPressEvent`. `TKeyPressEvent` defines two arguments: one to indicate which object generated the event, and one to indicate which key was pressed:

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char &Key);
```

Normally, the `Key` parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an edit control. In that case, the user could define the following handler for keystrokes:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

You can also use arguments passed by reference to let the user override the default handling.

Event Types Are closure Types (C++)

Because an event is a pointer to an event handler, the type of the event property must be a closure type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

C++Builder defines closures for all its standard events. When you create your own events, you can use an existing closure if that is appropriate, or define one of your own.

Event Handlers Are Optional

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event.)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most

cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. The components in the class library have events written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

Implementing the Standard Events

The controls that come with the component library inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying Standard Events

There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed, graphical, or custom, inherit these events. The following events are available in all controls:

- *OnClick*
- *OnDblClick*
- *OnDragDrop*
- *OnDragOver*
- *OnEndDrag*
- *OnMouseMove*
- *OnMouseDown*
- *OnMouseUp*

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

Standard events for standard controls

In addition to the events common to all controls, standard windowed controls (those that descend from *TWinControl*) have the following events:

- *OnEnter*
- *OnKeyPress*

- *OnKeyDown*
- *OnKeyUp*
- *OnExit*

Like the standard events in *TControl*, the windowed control events have corresponding methods. The standard key events listed above respond to all normal keystrokes.

Note: To respond to special keystrokes (such as the Alt key), however, you must respond to the WM_GETDLGCODE or CM_WANTSPECIALKEYS message from Windows. See Handling messages and system notifications for information on writing message handlers.

Making Events Visible

The declarations of the standard events in *TControl* and *TWinControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either **public** or **published**.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

Changing the Standard Event Handling

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

Defining Your Own Events

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the Event

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN*

message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

However, some events are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Two Kinds of Events

There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code.

You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

Defining the Handler Type

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

Declaring the Event

Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in Delphi begin with "On." This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Note: The main exception to this rule is that many events that occur before and after some occurrence begin with "Before" and "After."

Calling the Event

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid.
- Users can override default handling.

Empty Handlers Must Be Valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

Users Can Override Default Handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

Creating methods

Creating Methods: Overview

Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow. These guidelines include:

- Avoiding dependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi and are accessible at design time.

Avoiding Interdependencies

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

Naming Methods

Delphi imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs. A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.
- Function names should reflect the nature of what they return.

Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting Methods

All parts of classes, including fields, methods, and properties, have a level of protection or "visibility," as explained in Controlling access. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

Methods That Should Be Public

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

Note: Constructors and destructors should always be **public**.

Methods That Should Be Protected

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

Abstract Methods

Sometimes a method is declared as **abstract** in a Delphi component. In the component library, abstract methods usually occur in classes whose names begin with "custom," such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

Making Methods Virtual

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

Declaring Methods

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, do the following:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

Example of Declaring Methods

The following code shows a component that defines two new methods, one protected method and one public virtual method.

C++

This is the interface definition in the .H file:

```
[C++]
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
  .
  .
  .
};
```

This is the code in the .CPP file of the unit that implements the methods:

```
[C++]
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
}
int __fastcall TSampleComponent::CalculateArea()
{
  return Width * Height;
}
```

Delphi

```
[Delphi]
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                              { declare protected static method }
  public
    function CalculateArea: Integer; virtual;            { declare public virtual method }
  end;
.
.
.
implementation
.
.
.
procedure TSampleComponent.MakeBigger;                          { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;
function TSampleComponent.CalculateArea: Integer;              { implement second method }
begin
  Result := Width * Height;
end;
```

Using graphics in components

Using Graphics in Components: Overview

Windows provides a powerful graphics device interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources.

As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier.

The topics in this section include:

- Overview of graphics
- Using the canvas
- Working with pictures
- Off-screen bitmaps
- Responding to changes

Overview of Graphics

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it.

Instead of forcing you to deal with graphics at a detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas ensures that it has a valid device context, and releases the context when you are not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles, thanks to a font cache.

Using the Canvas

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and in the component library, provides low-level access to the Windows GDI.

The following table summarizes the capabilities of the canvas.

Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Manipulating pixels	*Pixels* property.
	Copying and merging images	*Draw, StretchDraw, BrushCopy,* and *CopyRect* methods; *CopyMode* property
Low	Calling Windows GDI functions	*Handle* property

Working with Pictures

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes

Using a Picture, Graphic, or Canvas

There are three kinds of classes in Delphi that deal with graphics:

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can

access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. (The only standard graphic that has a canvas is *TBitmap*.) Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and Storing Graphics

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

Handling Palettes

For VCL components, when running on a palette-based device (typically, a 256-color video mode), Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette, but controls that contain "rich color" graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the foremost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the "real" palette. As windows move in front of one another, Windows continually realizes the palettes.

Note: Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. If you have a palette handle, however, Delphi controls can manage it for you.

Specifying a Palette for a Control

To specify a palette for a control, override the control's *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does these things for your application:

- It tells the application that your control's palette needs to be realized.
- It designates the palette to use for realization.

Responding to Palette Changes

If your VCL control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step further, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

Off-screen Bitmaps

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

Creating and Managing Off-screen Bitmaps

When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                                  { override the Paint method }
  end;
procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                        { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;                                  { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                             { destroy the bitmap object }
  end;
end;
```

Copying Bitmapped Images

Delphi provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

The following table summarizes the image-copying methods in canvas objects.

Image-copying methods

To create this effect	Call this method
Copy an entire graphic.	Draw
Copy and resize a graphic.	StretchDraw
Copy part of a canvas.	CopyRect
Copy a bitmap with raster operations.	BrushCopy (VCL)

Responding to Changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

Handling messages

Handling Messages and System Notifications: Overview

Components often need to respond to notifications from the underlying operating system. The operating system informs the application of occurrences such as what the user does with the mouse and keyboard. Some controls also generate notifications, such as the results from user actions such as selecting an item in a list box. The component library handles most of the common notifications already. It is possible, however, that you will need to write your own code for handling such notifications.

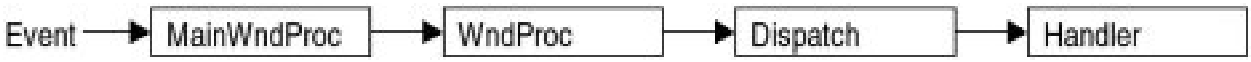
For VCL applications, notifications arrive in the form of *messages*. These messages can come from any source, including Windows, VCL components, and components you have defined. There are three aspects to working with messages:

- Understanding the message-handling system.
- Changing message handling.
- Creating new message handlers.

Understanding the message-handling system

All VCL classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. You should never need to alter this message-dispatch mechanism. All you will need to do is create message-handling methods. See the section Declaring a new message-handling method for more on this subject.

What's in a Windows Message?

A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. Other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as wParam and lParam, for *word parameter* and *long parameter*. Often, each parameter will contain more than one piece of information, and you see references to names such as lParamHi, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember or look up in the Windows APIs what each parameter contained. Now Microsoft has named the parameters. This so-called *message cracking* makes it much simpler to understand what information accompanies each message. For example, the parameters to the *WM_KEYDOWN* message are now called nVirtKey and lKeyData, which gives much more specific information than wParam and lParam.

For each type of message, Delphi defines a record type that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message structure, you do not have to worry about which word is which, because you refer to the parameters by the names XPos and YPos instead of lParamLo and lParamHi.

```
void MyKeyDownHandler(  HWND hwnd, UINT nVirtKey, BOOL fDown, int CRepeat, UINT flags )
{
  .
  .
  .
}
LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam )
{
  switch(  Message )
  {
    HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
    .
    .
    .
}
```

Dispatching Messages

When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

The VCL simplifies message dispatching in several ways:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only for messages you need to respond to specially.
- You can modify small parts of the message handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component does not have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

The VCL registers a method called MainWndProc as the window procedure for each type of component in an application. MainWndProc contains an exception-handling block, passing the message structure from Windows to

a virtual method called WndProc and handling any exceptions by calling the application class's HandleException method.

MainWndProc is a nonvirtual method that contains no special handling for any particular messages. Customizations take place in WndProc, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so the WndProc method of TWinControl passes along keyboard events only if the component is not being dragged. Ultimately, WndProc calls Dispatch, a nonvirtual method inherited from TObject, which determines which method to call to handle the message.

Dispatch uses the Msg field of the message structure to determine how to dispatch a particular message. If the component defines a handler for that particular message, Dispatch calls the method. If the component does not define a handler for that message, Dispatch calls DefaultHandler.

Changing Message Handling

Before changing the message handling of your components, make sure that is what you really want to do. The VCL translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change message handling in VCL components, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

Overriding the Handler Method

To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the **override** directive; you must use the **message** directive and a matching message index.

Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. For clarity, however, it is best to follow the convention of naming message-handling methods after the messages they handle.

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parameter1, parameter2, parameter3)
END_MESSAGE_MAP
```

Using Message Parameters

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the Result field for the message: the value returned by the SendMessage call that sends the message.

Because the type of the Message parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for some reason you need to refer to the message parameters by their old-style names (WParam, LParam, and so on), you can typecast Message to the generic type TMessage, which uses those parameter names.

Trapping Messages

Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, you override the virtual method WndProc.

For VCL components, the WndProc method screens messages before passing them to the Dispatch method, which in turn determines which method gets to handle the message. By overriding WndProc, your component gets a chance to filter out messages before dispatching them. An override of WndProc for a control derived from TWinControl looks like this:

```
[Delphi]
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);
end;
```

```
[C++]
void __fastcall TMyControl::WndProc(TMessage& Message)
{
   // tests to determine whether to continue processing
if(Message.Msg != WM_LBUTTONDOWN)

}
```

The TControl component defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding WndProc helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

The WndProc Method

Note: This information is applicable when writing VCL components only.

Here is part of the WndProc method for TControl, for example:

```
procedure TControl.WndProc(var Message: TMessage);
begin
  .
  .
  .
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                    { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      .                                                     { handle others normally }
      .
      .
    end;
.                                                       { otherwise process normally }
.
.
end;
```

Creating New Message Handlers

Because the VCL provides handlers for most common messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has three aspects:

- Defining your own messages.
- Declaring a new message-handling method.
- Sending messages.

Defining Your Own Messages

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard messages and notification of state changes. You can define your own messages in the VCL.

Defining a message is a two-step process. The steps are:

1. Declaring a message identifier.
2. Declaring a message-record type.

Declaring a Message Identifier

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant *WM_APP* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_APP*.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages unit to see which messages Windows already defines for that control.

Declaring a Message-structure Type

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message-record is the type of the parameter passed to the message-handling method. If you do not use the message's parameters, or if you want to use the old-style parameter notation (wParam, lParam, and so on), you can use the default message-record, TMessage.

To declare a message-record type, follow these conventions:

1. Name the record type after the message, preceded by a *T*.
2. Call the first field in the record Msg, of type TMsgParam.
3. Define the next two bytes to correspond to the Word parameter, and the next two bytes as unused.

   Or

   Define the next four bytes to correspond to the Longint parameter.
4. Add a final field called Result, of type Longint.

Declaring a New Message-handling Method

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that is not already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

1. Declare the method in a **protected** part of the component's class declaration.
2. Make the method a procedure.
3. Name the method after the message it handles, but without any underline characters.
4. Pass a single **var** parameter called Message, of the type of the message record.
5. Within the message method implementation, write code for any handling specific to the component.
6. Call the inherited message handler.

Sending Messages

Typically, an application sends message to send notifications of state changes or to broadcast information. Your component can broadcast messages to all the controls in a form, send messages to a particular control (or to the application itself), or even send messages to itself.

There are several different ways to send a Windows message. Which method you use depends on why you are sending the message. The following topics describe the different ways to send Windows messages:

- Broadcasting a message to all controls in a form.
- Calling a control's message handler directly.
- Sending a message using the Windows message queue.
- Sending a message that does not execute immediately.

Broadcasting a Message to All Controls in a Form

When your component changes global settings that affect all of the controls in a form or other container, you may want to send a message to those controls so that they can update themselves appropriately. Not every control may need to respond to the notification, but by broadcasting the message, you can inform all controls that know how to respond and allow the other controls to ignore the message.

To broadcast a message to all the controls in another control, use the Broadcast method. Before you broadcast a message, you fill out a message record with the information you want to convey.

```
[Delphi]
var
Msg: TMessage;
begin
  Msg.Msg := MY_MYCUSTOMMESSAGE;
  Msg.WParam := 0;
  Msg.LParam := Longint(Self);
  Msg.Result := 0;
```

```
[C++]
TMessage Msg;
Msg.Msg = MY_MYCUSTOMMESSAGE;
Msg.WParam = 0;
Msg.LParam = (int)(this);
Msg.Result = 0;
```

Then, pass this message record to the parent of all the controls you want to notify. This can be any control in the application. For example, it can be the parent of the control you are writing:

```
[Delphi]
Parent.Broadcast(Msg);
```

```
[C++]
Parent->Broadcast(Msg);
```

It can be the form that contains your control:

```
[Delphi]
GetParentForm(self).Broadcast(Msg);
```

```
[C++]
GetParentForm(this)->Broadcast(Msg);
```

It can be the active form:

```
[Delphi]
Screen.ActiveForm.Broadcast(Msg);
```

```
[C++]
Screen->ActiveForm->Broadcast(Msg);
```

It can even be all the forms in your application:

```
[Delphi]
for I:= 0 to Screen.FormCount - 1 do
  Screen.Forms[I].Broadcast(Msg);
```

```
[C++]
for (int i = 0; i < Screen->FormCount; i++)
  Screen->Forms[i]->Broadcast(Msg);
```

Calling a Control's Message Handler Directly

Sometimes there is only a single control that needs to respond to your message. If you know the control that should receive your message, the simplest and most straightforward way to send the message is to call the control's Perform method.

There are two main reasons why you call a control's *Perform* method:

- You want to trigger the same response that the control makes to a standard Windows (or other) message. For example, when a grid control receives a keystroke message, it creates an inline edit control and then sends the keystroke message on to the edit control.
- You may know what control you want to notify, but not know what type of control it is. Because you don't know the type of the target control, you cannot use any of its specialized methods, but all controls have message-handling capabilities so you can always send a message. If the control has a message handler for the message you send, it will respond appropriately. Otherwise, it will ignore the message you send and return 0.

To call the *Perform* method, you do not need to create a message record. You need only pass the message identifier, WParam, and LParam as parameters. Perform returns the message result.

Sending a Message Using the Windows Message Queue

In a multithreaded application, you can't just call the *Perform* method because the target control is in a different thread than the one that is executing. However, by using the Windows message queue, you can safely communicate with other threads. Message handling always occurs in the main VCL thread, but you can send a message using the Windows message queue from any thread in the application. A call to *SendMessage* is synchronous. That is, *SendMessage* does not return until the target control has handled the message, even if it is in another thread.

Use the Windows API call, *SendMessage*, to send a message to a control using the Windows message queue. *SendMessage* takes the same parameters as the *Perform* method, except that you must identify the target control by passing its Window handle. Thus, instead of writing

```
[Delphi]
MsgResult := TargetControl.Perform(MY_MYMESSAGE, 0, 0);
```

```
[C++]
MsgResult = TargetControl->Perform(MY_MYMESSAGE, 0, 0);
```

you would write

```
[Delphi]
MsgResult := SendMessage(TargetControl.Handle, MYMESSAGE, 0, 0);
```

```
[C++]
MsgResult = SendMessage(TargetControl->Handle, MYMESSAGE, 0, 0);
```

For more information on the SendMessage function, see the Microsoft MSDN documentation. For more information on writing multiple threads that may be executing simultaneously, see Coordinating threads.

Sending a Message That Does Not Execute Immediately

There are times you may want to send a message but you do not know whether it is safe for the target of the message to execute right away. For example, if the code that sends a message is called from an event handler on the target control, you may want to make sure that the event handler has finished executing before the control executes your message. You can handle this situation as long as you do not need to know the message result.

Use the Windows API call, *PostMessage*, to send a message to a control but allow the control to wait until it has finished any other messages before it handles yours. *PostMessage* takes exactly the same parameters as *SendMessage*.

For more information on the PostMessage function, see the Microsoft MSDN documentation.

Responding to Signals

The underlying widget layer emits a variety of signals, each of which represents a different type of notification. These signals include system events (the event signal) as well as notifications that are specific to the widget that generates them. For example, all widgets generate a destroyed signal when the widget is freed, trackbar widgets generate a *valueChanged* signal, header controls generate a *sectionClicked* signal, and so on.

Each CLX component responds to signals from its underlying widget by assigning a method as the handler for the signal. It does this using a special hook object that is associated with the underlying widget. The hook object is a lightweight object that is really just a collection of method pointers, each method pointer specific to a particular signal. When a method of the CLX component has been assigned to the hook object as the handler for a specific signal, then every time the widget generates the specific signal, the method on the CLX component gets called.

Note: The methods for each hook object are declared in the Qt unit. The methods are flattened into global routines with names that reflect the hook object to which they belong. For example, all methods on the hook object associated with the application widget (QApplication) begin with 'QApplication_hook.' This flattening is necessary so that the Delphi CLX object can access the methods of the C++ hook object.

Assigning Custom Signal Handlers

Many CLX controls already assign methods to handle signals from the underlying widget. Typically, these methods are private and not virtual. Thus, if you want to write your own method to respond to a signal, you must assign your own method to the hook object associated with your widget. To do this, override the *HookEvents* method.

Note: If the signal to which you want to respond is a system event notification, you must not use an override of the HookEvents method. For details on how to respond to system events, see Responding to system events.

In your override of the *HookEvents* method, declare a variable of type *TMethod*.

Then for each method you want to assign to the hook object as a signal handler, do the following:

1. Initialize the variable of type *TMethod* to represent a method handler for the signal.
2. Assign this variable to the hook object. You can access the hook object using the *Hooks* property that your component inherits from *THandleComponent* or *TWidgetControl*.

In your override, always call the inherited *HookEvents* method so that the signal handlers that base classes assign are also hooked up.

The following code is the *HookEvents* method of *TTrackBar*. It illustrates how to override the *HookEvents* method to add custom signal handlers.

```
[Delphi]
procedure TTrackBar.HookEvents;
var
        Method: TMethod;
begin
  // initialize Method to represent a handler for the QSlider valueChanged signal
  // ValueChangedHook is a method of TTrackBar that responds to the signal.
  QSlider_valueChanged_Event(Method) := ValueChangedHook;
  // Assign Method to the hook object. Note that you can cast Hooks to the
  // type of hook object associated with the underlying widget.
  QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
  // Repeat the process for the sliderMoved event:
  QSlider_sliderMoved_Event(Method) := ValueChangedHook;
```

```
  QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
  // Call the inherited method so that inherited signal handlers are hooked up:
  inherited HookEvents;
end;
```

```
[C++]
virtual void __fastcall TTrackBar::HookEvents(void)
{
  TMethod Method;
  // initialize Method to represent a handler for the QSlider valueChanged signal
  // ValueChangedHook is a method of TTrackBar that responds to the signal.
  QSlider_valueChanged_Event(Method) = @ValueChangedHook;
  // Assign Method to the hook object. Note that you can cast Hooks to the
  // type of hook object associated with the underlying widget.
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
  // Repeat the process for the sliderMoved event:
  QSlider_sliderMoved_Event(Method) := @ValueChangedHook;
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
  // Call the inherited method so that inherited signal handlers are hooked up:
  TWidgetControl::HookEvents();
}
```

Responding to System Events

When the widget layer receives an event notification from the operating system, it generates a special event object (*QEvent* or one of its descendants) to represent the event. The event object contains read-only information about the event that occurred. The type of the event object indicates the type of event that occurred.

The widget layer notifies your CLX component of system events using a special signal of type event. It passes the *QEvent* object to the signal handler for the event. The processing of the event signal is a bit more complicated than processing other signals because it goes first to the application object. This means an application has two opportunities to respond to a system event: once at the application level (*TApplication*) and once at the level of the individual component (your *TWidgetControl* or *THandleComponent* descendant.) All of these classes (*TApplication*, *TWidgetControl*, and *THandleComponent*) already assign a signal handler for the event signal from the widget layer. That is, all system events are automatically directed to the *EventFilter* method, which plays a role similar to the *WndProc* method on VCL controls.

EventFilter handles most of the commonly used system notifications, translating them into the events that are introduced by your component's base classes. Thus, for example, the *EventFilter* method of *TWidgetControl* responds to mouse events (*QMouseEvent*) by generating the *OnMouseDown*, *OnMouseMove*, and *OnMouseUp* events, to keyboard events (*QKeyEvent*) by generating the *OnKeyDown*, *OnKeyPress*, *OnKeyString*, and *OnKeyUp* events, and so on.

The following topics describe how to customize the way your control works with system events:

- Commonly used events
- Overriding the EventFilter method
- Generating Qt events

Commonly Used Events

The *EventFilter* method of *TWidgetControl* handles many of the common system notifications by calling on protected methods that are introduced in *TControl* or *TWidgetControl*. Most of these methods are virtual or dynamic, so that you can override them when writing your own components and implement your own responses to the system event.

When overriding these methods, you do not need to worry about working with the event object or (in most cases) any of the other objects in the underlying widget layer.

When you want your CLX component to respond to system notifications, it is a good idea to first check whether there is a protected method that already responds to the notification. You can check the documentation for *TControl* or *TWidgetControl* (and any other base classes from which you derive your component) to see if there is a protected method that responds to the event in which you are interested. The following table lists many of the most commonly used protected methods from *TControl* and *TWidgetControl* that you can use.

TWidgetControl protected methods for responding to system notifications

Method	Description
BeginAutoDrag	Called when the user clicks the left mouse button if the control has a *DragMode* of *dmAutomatic*.
Click	Called when the user releases the mouse button over the control.
DblClick	Called when the user double-clicks with the mouse over the control.
DoMouseWheel	Called when the user rotates the mouse wheel.
DragOver	Called when the user drags the mouse cursor over the control.
KeyDown	Called when the user presses a key while the control has focus.
KeyPress	Called after *KeyDown* if *KeyDown* does not handle the keystroke.
KeyString	Called when the user enters a keystroke when the system uses a multibyte character system.
KeyUp	Called when the user releases a key while the control has focus.
MouseDown	Called when the user clicks the mouse button over the control.
MouseMove	Called when the user moves the mouse cursor over the control.
MouseUp	Called when the user releases the mouse button over the control.
PaintRequest	Called when the system needs to repaint the control.
WidgetDestroyed	Called when a widget underlying a control is destroyed.

In the override, call the inherited method so that any default processes still take place.

Note: In addition to the methods that respond to system events, controls include a number of similar methods that originate with *TControl* or *TWidgetControl* to notify the control of various events.Although these do not respond to system events, they perform the same task as many Windows messages that are sent to VCL controls. The following table lists some of these methods.

TWidgetControl protected methods for responding to events from controls

Method	Description
BoundsChanged	Called when the control is resized.
ColorChanged	Called when the color of the control changes.
CursorChanged	Called when the cursor changes shape. The mouse cursor assumes this shape when it's over this widget.
EnabledChanged	Called when an application changes the enabled state of a window or control.
FontChanged	Called when the collection of font resources changes.
PaletteChanged	Called when the widget's palette changes.
ShowHintChanged	Called when Help hints are displayed or hidden on a control.
StyleChanged	Called when the window or control's GUI styles change.
TabStopChanged	Called when the tab order on the form changes.
TextChanged	Called when the control's text changes.

VisibleChanged	Called when a control is hidden or shown.

Overriding the EventFilter Method

If you want to respond to an event notification and there is no protected method for that event that you can override, you can override the *EventFilter* method itself. In your override, check the type of the *Event* parameter of the *EventFilter* method, and perform your special processing when it represents the type of notification to which you want to respond. You can prevent further processing of the event notification by having your *EventFilter* method return *True*.

Note: See the Qt documentation from TrollTech for details about the different types of *QEvent* objects.

The following code is the *EventFilter* method on *TCustomControl*. It illustrates how to obtain the event type from the *QEvent* object when overriding *EventFilter*. Note that, although it is not shown here, you can cast the QEvent object to an appropriate specialized *QEvent* descendant (such as *QMouseEvent*) once you have identified the event type.

```
[Delphi]
function TCustomControl.EventFilter(Sender: QObjectH; Event: QEventH): Boolean;
begin
    Result := inherited EventFilter(Sender, Event);
    case QEvent_type(Event) of
        QEventType_Resize,
        QEventType_FocusIn,
        QEventType_FocusOut:
            UpdateMask;
    end;
end;
```

```
[C++]
virtual bool __fastcall TCustomControl::EventFilter(Qt::QObjectH* Sender, Qt::QEventH*
Event)
{
  bool retval = TWidgetControl::EventFilter(Sender, Event);
  switch (QEvent_type(Event))
  {
    case QEventType_Resize:
    case QEventType_FocusIn:
    case QEventType_FocusOut:
      UpdateMask();
  }
  return retval;
}
```

Generating Qt Events

Similar to the way a VCL control can define and send custom Windows messages, you can make your CLX control define and generate Qt system events. The first step is to define a unique ID for the event (similar to the way you must define a message ID when defining a custom Windows message):

```
[Delphi]
const
  MyEvent_ID = Integer(QCLXEventType_ClxUser) + 50;
```

```
[C++]
static const MyEvent_ID = (int) QCLXEventType_ClxUser + 50;
```

In the code where you want to generate the event, use the *QCustomEvent_create* function (declared in the Qt unit) to create an event object with your new event ID. An optional second parameter lets you supply the event object with a data value that is a pointer to information you want to associate with the event:

```
[Delphi]
var
  MyEvent: QCustomEventH;
begin
  MyEvent := QCustomEvent_create(MyEvent_ID, self);
```

```
[C++]
QCustomEventH *MyEvent = QCustomEvent_create(MyEvent_ID, this);
```

Once you have created the event object, you can post it by calling the *QApplication_postEvent* method:

```
[Delphi]
QApplication_postEvent(Application.Handle, MyEvent);
```

```
[C++]
QApplication_postEvent(Application->Handle, MyEvent);
```

For any component to respond to this notification, it need only override its *EventFilter* method, checking for an event type of *MyEvent_ID*. The *EventFilter* method can retrieve the data you supplied to the constructor by calling the *QCustomEvent_data* method that is declared in the Qt unit.

Making components available at design time

Making Components Available at Design Time: Overview

Making your components available at design time requires several steps:

- Registering components
- Providing Help for your component
- Adding property editors
- Adding component editors
- Compiling components into packages

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see Installing component packages.

Registering Components

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Tool palette.

Note: If you create your component by choosing **Component** ▸ **New Component** in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

Declaring the Register Procedure

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Delphi) its name is case-sensitive.

Note: Although Delphi is a case insensitive language, the Register procedure is case sensitive and must be spelled with an uppercase R.

The following code shows the outline of a simple unit that creates and registers new components:

```
[Delphi]
unit MyBtns;
interface
type
  ...                                                { declare your component types here }
procedure Register;                                { this must appear in the interface section }
implementation
  ...                                                { component implementation goes here }
procedure Register;
begin
  ...                                                          { register the components }
end;
end.
```

```
[C++]
namespace Newcomp
{
void __fastcall PACKAGE Register()
{
}
}
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Tool palette. If the unit contains several components, you can register them all in one step.

Writing the Register Procedure

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Tool palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each category of the Tool palette to which you want to add components. *RegisterComponents* involves three important things:

1. Specifying the components.
2. Specifying the palette page.
3. Using the RegisterComponents function.

Specifying the Components

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
[Delphi]
RegisterComponents('Miscellaneous', [TMyComponent]);
```

```
[C++]
TMetaClass classes[1] = {__classid(TNewComponent)};
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
[Delphi]
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);            { two on this page... }
  RegisterComponents('Assorted', [TThird]);                         { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
```

```
[C++]
TMetaClass classes[2] =
{__classid(TNewComponent), __classid(TAnotherComponent)};
```

```
[C++]

//Another way to add a component to the array
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

Specifying the Palette Page

The palette category name is a string. If the name you give for the palette category does not already exist, Delphi creates a new category with that name. Delphi stores the names of the standard categories in string-list resources so that international versions of the product can name the categories in their native languages. If you want to install a component on one of the standard categories, you should obtain the string for the category name by calling the *LoadStr* function, passing the constant representing the string resource for that category, such as *srSystem* for the System category.

Using the RegisterComponents Function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Tool palette category and the array of component classes.

Set the Page parameter to the name of the category on the Tool palette where the components should appear. If the named category already exists, the components are added to that category. If the named category does not exist, Delphi creates a new palette category with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the Tool palette.

```
[Delphi]
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);                    {add to system
category}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);                    { new category}
end;
```

```
[C++]
namespace Newcomp
{
void __fastcall PACKAGE Register()
{
TMetaClass classes[1] = {__classid(TMyComponent)};
RegisterComponents("Miscellaneous", classes, 0);
}
}
```

```
[C++]
namespace Mycomps
{
 void __fastcall PACKAGE Register()
{
// declares an array that holds two components
TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
// adds a new palette page with the two components in the classes1 array
RegisterComponents("Miscellaneous", classes1, 1);
// declares a second array
TMetaClass classes2[1];
// assigns a component to be the first element in the array
classes2[0]  = __classid(TThird);
// adds the component in the classes2 array to the Samples page
RegisterComponents("Samples", classes2, 0);
}
}
```

Providing Help for Your Component

When you select a standard component on a form, or a property or event in the Object Inspector, you can press F1 to get Help on that item. You can provide developers with the same kind of documentation for your components if you create the appropriate Help files.

You can provide a small Help file to describe your components, and your Help file becomes part of the user's overall Delphi Help system.

See the section Creating the Help file for information on how to compose the Help file for use with a component.

Creating the Help File

You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online Help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop.

Composing Help files for components consists of the steps:

- Creating the entries.

- Making component Help context-sensitive.

Creating the Entries

To make your component's Help integrate seamlessly with the Help for the rest of the components in the library, observe the following conventions:

Each component should have a Help topic:

The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing F1. For an example of a component topic, place any component on a form and press F1.

The component topic must have a # footnote with a value unique to the topic. The # footnote uniquely identifies each topic by the Help system.

The component topic should have a K footnote for keyword searching in the Help system Index that includes the name of the component class. For example, the keyword footnote for the *TMemo* component is "TMemo."

The component topic should also have a $ footnote that provides the title of the topic. The title appears in the Topics Found dialog box, the Bookmark dialog box, and the History window.

Each component should include the following secondary navigational topics:

- A hierarchy topic with links to every ancestor of the component in the component hierarchy.
- A list of all properties available in the component, with links to entries describing those properties.
- A list of all events available in the component, with links to entries describing those events.
- A list of methods available in the component, with links to entries describing those methods.

Links to object classes, properties, methods, or events in the Delphi Help system can be made using Alinks. When linking to an object class, the Alink uses the class name of the object, followed by an underscore and the string "object". For example, to link to the *TCustomPanel* object, use the following:

```
!AL(TCustomPanel_object,1)
```

When linking to a property, method, or event, precede the name of the property, method, or event by the name of the object that implements it and an underscore. For example, to link to the *Text* property which is implemented by *TControl*, use the following:

```
!AL(TControl_Text,1)
```

To see an example of the secondary navigation topics, display the Help for any component and click on the links labeled hierarchy, properties, methods, or events.

Each property, method, and event that is declared within the component should have a topic:

A property, event, or method topic should show the declaration of the item and describe its use. Application developers see these topics either by highlighting the item in the Object Inspector and pressing F1 or by placing the cursor in the Code editor on the name of the item and pressing F1. To see an example of a property topic, select any item in the Object Inspector and press F1.

The property, event, and method topics should include a K footnote that lists the name of the property, method, or event, and its name in combination with the name of the component. Thus, the *Text* property of *TControl* has the following K footnote:

```
Text,TControl;TControl,Text;Text,
```

The property, method, and event topics should also include a $ footnote that indicates the title of the topic, such as TControl.Text.

All of these topics should have a topic ID that is unique to the topic, entered as a # footnote.

Making Component Help Context-sensitive

Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses F1, or when a property or event is selected in the Object Inspector and the user presses F1. The A footnotes must follow certain naming conventions:

If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:

```
ComponentClass_Object;ComponentClass
```

where *ComponentClass* is the name of the component class.

If the Help topic is for a property or event, the A footnote consists of three entries separated by semicolons using this syntax:

```
ComponentClass_Element;Element_Type;Element
```

where *ComponentClass* is the name of the component class, *Element* is the name of the property, method, or event, and *Type* is the either Property, Method, or Event

For example, for a property named *BackgroundColor* of a component named *TMyGrid*, the A footnote is

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

Adding Property Editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires these five steps:

1. Deriving a property-editor class.
2. Editing the property as text.
3. Editing the property as a whole.
4. Specifying editor attributes.
5. Registering the property editor.

Deriving a Property-editor Class

Both the component library define several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in the table below. The classes in the DesignEditors unit can be used for VCL applications.

Note: All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

The list in the table below is not complete. The VCLEditors unit also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

Setting the Property Value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the Set methods.

Editing the Property as a Whole

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the Edit method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

Specifying Editor Attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.
paVolatileSubProperties	GetProperties	The Object Inspector re-fetches the values of all subproperties any time the property value changes.

paReference	GetComponentValue	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

Registering the Property Editor

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit—this is always a call to the built-in function *TypeInfo*, such as TypeInfo(TMyComponent)`__typeinfo(TMyComponent)`.
- The type of the component to which this editor applies—if this parameter is **nil**, the editor applies to all properties of the given type.
- The name of the property—this parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Property Categories

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

- Registering one property at a time
- Registering multiple properties at once
- Specifying property categories
- Using the IsPropertyInCategory function

Registering One Property at a Time

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
[Delphi]
RegisterPropertyInCategory('Visual', 'AutoSize');
```

```
[C++]
RegisterPropertyInCategory("Visual", "AutoSize");
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the 'Help and Hints' category) a property named "HelpContext" of a component of the custom class *TMyButton*.

```
[Delphi]
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

```
[C++]
RegisterPropertyInCategory("Help and Hints", __classid(TMyButton), "HelpContext");
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, Integer.

```
[Delphi]
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

```
[C++]
RegisterPropertyInCategory("Visual", typeid(TArrangement));
```

The final variation uses both the property's type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, "Pattern."

```
[Delphi]
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

```
[C++]
RegisterPropertyInCategory("Visual", typeid(TBitmap), "Pattern");
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Registering Multiple Properties at Once

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable.'

```
[Delphi]
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

```
[C++]
RegisterPropertiesInCategory("Localizable", ARRAYOFCONST("Text", __typeinfo(TEdit)));
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types. For example, the following code registers a number of properties into the 'Help and Hints' category for all components:

```
[Delphi]
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
```

```
[C++]
RegisterPropertyInCategory("Help and Hints", __classid(TComponent), ARRAYOFCONST
("HelpContext", "Hint", "ParentShowHint"));
```

The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
[Delphi]
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

```
[C++]
RegisterPropertiesInCategory("Localizable", __typeinfo(TStrings), ARRAYOFCONST("Lines",
"Commands"));
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Specifying Property Categories

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in the following table:

Property categories

Category	Purpose
Action	Properties related to runtime actions; the *Enabled* and *Hint* properties of *TEdit* are in this category.
Database	Properties related to database operations; the *DatabaseName* and *SQL* properties of *TQuery* are in this category.
Drag, Drop, and Docking	Properties related to drag-and-drop and docking operations; the *DragCursor* and *DragKind* properties of *TImage* are in this category.
Help and Hints	Properties related to using online Help or hints; the *HelpContext* and *Hint* properties of *TMemo* are in this category.

Layout	Properties related to the visual display of a control at design-time; the *Top* and *Left* properties of *TDBEdit* are in this category.
Legacy	Properties related to obsolete operations; the *Ctl3D* and *ParentCtl3D* properties of *TComboBox* are in this category.
Linkage	Properties related to associating or linking one component to another; the *DataSet* property of *TDataSource* is in this category.
Locale	Properties related to international locales; the *BiDiMode* and *ParentBiDiMode* properties of *TMainMenu* are in this category.
Localizable	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.
Visual	Properties related to the visual display of a control at runtime; the *Align* and *Visible* properties of *TScrollBox* are in this category.
Input	Properties related to the input of data (need not be related to database operations); the *Enabled* and *ReadOnly* properties of *TEdit* are in this category.
Miscellaneous	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.

Using the IsPropertyInCategory Function

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must belong to a *TCustomEdit* descendant, have the name "Text," and be in the property category 'Localizable'.

```
[Delphi]
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

```
[C++]
IsItThere = IsPropertyInCategory("Localizable", __classid(TCustomEdit), "Text");
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must be a *TCustomEdit* descendant, have the name "Text", and be in the property category 'Localizable'.

```
[Delphi]
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
```

```
[C++]
IsItThere = IsPropertyInCategory("Localizable", "TCustomEdit", "Text");
```

Adding Component Editors

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

Adding Items to the Context Menu

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

Specifying Menu Items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
[Delphi]
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;
function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := '&DoThis ...';
    1: Result := 'Do&That';
  end;
end;
```

```
[C++]
int __fastcall TMyEditor::GetVerbCount(void)
{
return 2;
}
System::AnsiString __fastcall TMyEditor::GetVerb(int Index)
{
switch (Index)
{
case 0: return "&DoThis ..."; break;
case 1: return "Do&That"; break;
}
}
```

Note: Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

Implementing Commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

```
[Delphi]
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);       { instantiate the editor }
         if MySpecialDialog.Execute then;                 { if the user OKs the dialog... }
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the value }
         MySpecialDialog.Free;                                    { destroy the editor }
       end;
    1: That;                                                 { call the That method }
  end;
end;
```

```
[C++]
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
switch (Index)
{
case 0:
TMyDialog *MySpecialDialog = new TMyDialog();
MySpecialDialog->Execute();
((TMyComponent *)Component)->ThisProperty = MySpecialDialog->ReturnValue;
delete MySpecialDialog;
break;
case 1:
That();  // call the "That" method
break;
```

```
}
}
```

Changing the Double-click Behavior

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```
[Delphi]
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

```
[C++]
void __fastcall TMyEditor::Edit(void)
{
TFontDialog *pFontDlg = new TFontDialog(NULL);
pFontDlg->Execute();
((TMyComponent *)Component)->Font = pFontDlg->Font;
delete pFontDlg;
}
```

Note: If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```
[Delphi]

procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
```

```
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

```
[C++]
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
bool &Continue, bool &FreeEditor)
{
if (PropertyEditor->ClassNameIs("TMethodProperty") &&
CompareText(PropertyEditor->GetName, "OnSpecialEvent") == 0)
{
TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
}
}
```

Adding Clipboard Formats

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.

```
[Delphi]
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

```
[C++]
void __fastcall TMyComponentEditor::Copy(void)
{
WORD AFormat;
int AData;
HPALETTE APalette;
((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat, AData, APalette);
TClipboard *pClip = Clipboard(); // don't clear the clipboard!
pClip->SetAsHandle(AFormat, AData);
}
```

Registering the Component Editor

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
[Delphi]
RegisterComponentEditor(TMyComponent, TMyEditor);
```

```
[C++]
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```
[Delphi]

procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

```
[C++]
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {

    RegisterComponents("Miscellaneous", classes, 0);
    RegisterComponentEditor(classes[0], __classid(TMyEditor));
  }
}
```

Compiling Components into Packages

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Working with packages and components .

To create and compile a package, see Creating and editing packages . Put the source-code units for your custom components in the package's Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see Installing component packages.

Troubleshooting Custom Components (C++)

A common problem when registering and installing custom components is that the component does not appear in the list of components after the package is successfully installed.

The most common causes for components not appearing in the list or on the palette:

- Missing PACKAGE modifier on the `Register` function
- Missing PACKAGE modifier on the class
- Missing `#pragma package(smart_init)` in the C++ source file
- `Register` function is not found in a namespace with the same name as the source code module name.

- `Register` is not being successfully exported. Use tdump on the .BPL to look for the exported function:

tdump -ebpl mypack.bpl mypack.dmp

In the exports section of the dump, you should see the `Register` function (within the namespace) being exported.

Modifying an existing component

Modifying an Existing Component: Overview

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note: To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

Modifying an existing component takes only two steps:

- Creating and registering the component.
- Modifying the component class.

Creating and Registering the Component

You create every component the same way: you create a unit, derive a component class, register it, and install it on the Tool palette. This process is outlined in Creating a new component.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Tool palette.
- The resulting unit should look like this:

```
[Delphi]
unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
```

```
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

```
[C++]
#include <vcl.h>
#pragma hdrstop
#include "Yelmemo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TYellowMemo *)
{
  new TYellowMemo(NULL);
}
//---------------------------------------------------------------------------
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
: TMemo(Owner)
{
}
//---------------------------------------------------------------------------
namespace Yelmemo
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TYellowMemo)};
    RegisterComponents("Samples", classes, 0); //"Common Controls" in CLX applications
  }
}
```

```
[C++]
#ifndef YelMemoH
#define YelmemoH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TYellowMemo : public TMemo
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

Modifying the Component Object

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.dfm for VCL applications) file. Delphi stores only values that differ from the default, so it is important to perform both steps.

Overriding the Constructor

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note: When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see Overriding methods.

For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```
[Delphi]
type
  TWrapMemo = class(TMemo)
  public                                              { constructors are always public }
    constructor Create(AOwner: TComponent); override; { this syntax is always the same }
  end;
.
.
.
constructor TWrapMemo.Create(AOwner: TComponent);     { this goes after implementation }
begin
  inherited Create(AOwner);                           { ALWAYS do this first! }
  WordWrap := False;                                  { set the new desired value }
end;
```

```
[C++]
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner); // the constructor declaration
__published:
    __property Color;
};
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
  : TMemo(Owner)                                  // the constructor implementation
first...                                                 // ...calls the constructor for TMemo
{
  Color = clYellow;                              // colors the component yellow
}
```

Now you can install the new component on the Tool palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

Specifying the New Default Property Value

When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Making components available at design time.

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. You don't need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```
[Delphi]
type
  TWrapMemo = class(TMemo)
  .
  .
  .
  published
    property WordWrap default False;
  end;
```

```
[C++]

//header file
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
};
```

```
[C++]

//implmentation file
__fastcall TYellowMemo::TYellowMemo(TComponent* AOwner) : TMemo(AOwner)
{
  Color = clYellow;
  WordWrap = false;
}
```

```
[C++]

//header file with WordWrap as default value of false:
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
```

```
__published:
    __property Color = {default=clYellow};
    __property WordWrap = {default=false};
};
```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component's constructor. Redeclaring the default ensures that Delphi knows when to write *WordWrap* to the form file.

Creating a graphic component

Creating a Graphic Component

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need its own window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic control presented in the following topics is *TShape*, the shape component on the Additional page of the Tool palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. The following topics use the name *TSampleShape* and show you all the steps involved in creating the shape component:

- Creating and registering the component.
- Publishing inherited properties.
- Adding graphic capabilities.

Creating and Registering the Component

You create every component in the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component.

For this example, follow the general procedure for creating a component, with these specifics:

1. Call the component's unit *Shapes*.
2. Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
3. Register *TSampleShape* on the Samples category of the Tool palette.

The resulting unit should look like this:

```
[Delphi]
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
```

```
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

```
[C++]
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Shapes.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TSampleShape *)
{
  new TSampleShape(NULL);
}
//---------------------------------------------------------------------------
__fastcall TSampleShape::TGraphicControl(TComponent* Owner)
: TGraphicControl(Owner)
{
}
//---------------------------------------------------------------------------
namespace Shapes
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

```
[C++]
//---------------------------------------------------------------------------
#ifndef ShapesH
#define ShapesH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

Publishing Inherited Properties

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in Publishing inherited properties and Making events visible. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;        { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;        { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
private:
__published:
    __property DragCursor ;
    __property DragMode ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
};
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

Adding Graphic Capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

1. Determining what to draw.
2. Drawing the component image.

In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.

Determining What to Draw

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

1. Declaring the property type.
2. Declaring the property.
3. Writing the implementation method.

Creating properties is explained in more detail in Creating properties.

Declaring the Property Type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class's declaration.

```
[Delphi]
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { this is already there }
```

```
[C++]
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect, sstRoundSquare, sstEllipse,
sstCircle };
class PACKAGE TSampleShape : public TGraphicControl         // this is already there
```

You can now use this type to declare a new property in the class.

Declaring the Property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call.

Add the following declarations to *TSampleShape*:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;  { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
```

```
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
private:
    TSampleShapeType FShape;
    void __fastcall SetShape(TSampleShapeType Value);
__published:
    __property TSampleShapeType Shape = {read=FShape, write=SetShape, nodefault};
};
```

Now all that remains is to add the implementation of *SetShape*.

Writing the Implementation Method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
[Delphi]
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                          { ignore if this isn"t a change }
  begin
    FShape := Value;                               { store the new value }
    Invalidate;                                    { force a repaint with the new shape }
  end;
end;
```

```
[C++]
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value)             // ignore if this isn't a change
  {
     FShape = Value;               // store the new value
     Invalidate();                 // force a repaint with the new shape
  }
}
```

Overriding the Constructor and Destructor

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Modifying an existing component.

In this example, the shape control sets its size to a square 65 pixels on each side.

1 Add the overridden constructor to the declaration of the component class:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  public                                                { constructors are always public }
    constructor Create(AOwner: TComponent); override       { remember override directive }
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent *Owner);
};
```

2 Redeclare the *Height* and *Width* properties with their new default values:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  published
    property Height default 65;
    property Width default 65;
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
__published:
    __property Height;
    __property Width;
}
```

3 Write the new constructor in the **implementation** part of the unit:

```
[Delphi]
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);  { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

```
[C++]
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
```

```
  Width = 65;
  Height = 65;
}
```

Publishing the Pen and Brush

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires:

1. Declaring the class fields.
2. Declaring the access properties.
3. Initializing owned classes.
4. Setting owned classes' properties.

Declaring the Class Fields

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.

Add fields for a pen and brush to the shape control:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  private            { fields are nearly always private }
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    .
    .
    .
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
private:               // data members are always private
    TPen *FPen;        // a data member for the pen object
    TBrush *FBrush;    // a data member for the brush object
    .
    .
    .
};
```

Declaring the Access Properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  private                                              { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                          { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetBrush(TBrush *Value);
    void __fastcall SetPen(TPen *Value);
    .
    .
    .
__published:
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
};
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
[Delphi]
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                          { replace existing brush with parameter }
end;
procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                              { replace existing pen with parameter }
end;
```

```
[C++]
void __fastcall TSampleShape::SetBrush( TBrush* Value)
{
```

```
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen( TPen* Value)
{
  FPen->Assign(Value);
}
```

To directly assign the contents of *Value* to *FBrush*-

```
[Delphi]
  FBrush := Value;
```

```
[C++]
  FBrush = Value;
```

- would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

Initializing Owned Classes

If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1 Construct the pen and brush in the shape control constructor:

```
[Delphi]
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                    { construct the pen }
  FBrush := TBrush.Create;                               { construct the brush }
end;
```

```
[C++]
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();                          // construct the pen
  FPen = new TPen();                              // construct the brush
}
```

2 Add the overridden destructor to the declaration of the component class:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  public                                            { destructors are always public}
```

```
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                          { remember override directive }
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
public:                                                 // destructors are always public
    virtual __fastcall TSampleShape(TComponent* Owner);
    __fastcall ~TSampleShape();                         // the destructor
    .
    .
    .
};
```

3 Write the new destructor in the **implementation** part of the unit:

```
[Delphi]
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                               { destroy the brush object }
  inherited Destroy;                          { always call the inherited destructor, too }
end;
```

```
[C++]
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;                                          // delete the pen object
  delete FBrush;                                        // delete the brush object
}
```

Setting Owned Classes' Properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
.
.
.
implementation
```

```
.
.
.
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                      { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                              { construct the pen }
  FPen.OnChange := StyleChanged;                      { assign method to OnChange event }
  FBrush := TBrush.Create;                                        { construct the brush }
  FBrush.OnChange := StyleChanged;                    { assign method to OnChange event }
end;
procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;                                    { erase and repaint the component }
end;
```

```
[C++]

//header file
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
public:
    void __fastcall StyleChanged(TObject* Owner);
    .
    .
    .
};
```

```
[C++]

//implmentation file
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
```

```
[C++]

//also include StyleChanged method in the implementation file
void __fastcall TSampleShape::StyleChanged( TObject* Sender)
{
  Invalidate();         // repaints the component
}
```

With these changes, the component redraws to reflect changes to either the pen or the brush.

Drawing the Component Image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the Paint method requires two steps:

1. Add *Paint* to the component's declaration.
2. Write the *Paint* method in the **implementation** part of the unit.

For the shape control, add the following declaration to the class declaration:

```
[Delphi]
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  protected
    procedure Paint; override;
  .
  .
  .
  end;
```

```
[C++]
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
protected:
    virtual void __fastcall Paint();
    .
    .
    .
};
```

Then write the method in the **implementation** part of the unit:

```
[Delphi]
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                    { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
```

```
        Rectangle(0, 0, Width, Height);                    { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                              { draw round shapes }
    end;
  end;
end;
```

```
[C++]
void __fastcall TSampleShape::Paint()
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                                  // copy the component's pen
  Canvas->Brush = FBrush;                              // copy the component's brush
  W=Width;                                             // use the component width
  H=Height;                                            // use the component height
  X=Y=0;                                               // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)
  {
    case sstRectangle:                                  // draw rectangles and squares
    case sstSquare:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundRect:                                  // draw rounded rectangles and squares
    case sstRoundSquare:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                                     // draw circles and ellipses
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Paint is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

Refining the Shape Drawing

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

```
[Delphi]
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin  with Canvas do
```

```
  begin
    Pen := FPen;                                              { copy the component's pen }
    Brush := FBrush;                                        { copy the component's brush }
    W := Width;                                               { use the component width }
    H := Height;                                            { use the component height }
    if W < H then S := W else S := H;                { save smallest for circles/squares }
    case FShape of                                   { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                  { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                            { center these horizontally... }
          Y := (H - S) div 2;                                        { ...and vertically }
          W := S;                                  { use shortest dimension for width... }
          H := S;                                                { ...and for height }
        end;
    end;
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                    { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);           { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                               { draw round shapes }
    end;
  end;
end;
```

```
[C++]
void __fastcall TSampleShape::Paint(void)
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                          // copy the component's pen
  Canvas->Brush = FBrush;                      // copy the component's brush
  W=Width;                                     // use the component width
  H=Height;                                    // use the component height
  X=Y=0;                                       // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)                                // adjust height, width and position
  {
    case sstRectangle:
    case sstRoundRect:
    case sstEllipse:
      Y=X=0;                                 // origin is top-left for these shapes
      break;
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X= (W-S)/2;                             // center these horizontally
      Y= (H-S)/2;                             // and vertically
      break;
    default:
    break;
  }
```

```
  switch(FShape)
  {
    case sstSquare:                          // draw rectangles and squares
      W=H=S;                                 // use shortest dimension for width and height
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:                     // draw rounded rectangles and squares
      W=H=S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                          // draw circles and ellipses
      W=H=S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Customizing a grid

Customizing a Grid: Overview

The component library provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. The following topics describe how to create a small one month calendar from the basic grid component, *TCustomGrid*:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

In VCL applications, the resulting component is similar to the *TCalendar* component on the Samples category of the Tool palette. See Specifying the palette page.

Creating and registering the component

You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. Creating a new component.

For this example, follow the general procedure for creating a component, with these specifics:

1. Save the component's unit as *CalSamp*.
2. Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
3. Register *TSampleCalendar* on the Samples category of the Tool palette.

The resulting unit descending from *TCustomGrid* in a VCL application should look like this:

```
[Delphi]
unit CalSamp;
interface
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

```
[C++]
#include <vcl\vcl.h>
#pragma hdrstop
#include "CalSamp.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TSampleCalendar *ValidCtrCheck()
{
  return new TSampleCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Calsamp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0); //Use a different page in CLX applications
  }
}
```

```
[C++]
#ifndef CalSampH
#define CalSampH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
#include <vcl\grids.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

If you install the calendar component now, you will find that it appears on the Samples category. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note: While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in Filling in the cells

Publishing Inherited Properties

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component's declaration.

For the calendar control, publish the following properties and events, as shown here:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;  { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { publish events }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

```
[C++]
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
__published:
    __property Align ;                    // publish properties
    __property BorderStyle ;
    __property Color ;
    __property Font ;
    __property GridLineWidth ;
    __property ParentColor ;
    __property ParentFont ;
    __property OnClick ;                   // publish events
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnKeyDown ;
    __property OnKeyPress ;
```

```
    __property OnKeyUp ;
};
```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Tool palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

Changing Initial Values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

Remember that you need to add the constructor to the **public** part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor. Then add the StdCtrls unit to the **uses** clause.

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  public
    constructor Create(AOwner: TComponent); override;
  .
  .
  .
  end;
.
.
.
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                    { call inherited constructor }
  ColCount := 7;                                                  { always seven days/week }
  RowCount := 7;                                          { always six weeks plus the headings }
  FixedCols := 0;                                                          { no row labels }
  FixedRows := 1;                                                { one row for day names }
  ScrollBars := ssNone;                                             { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {disable range selection}
end;
```

```
[C++]
//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
protected:
    virtual void __fastcall DrawCell(int ACol, int ARow, const Windows::TRect &Rect,
      TGridDrawState AState);
    .
    .
    .
public:
```

```
    __fastcall TSampleCalendar(TComponent *Owner);   // the added constructor
    .
    .
    .
};
```

```
[C++]

//implementation file
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = (Options >> goRangeSelect) << goDrawFocusSelected;
}
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect
  &ARect, TGridDrawState AState)
{
}
```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the Cells

Note: When a user or application changes the size of a window or control, Windows sends a message called *WM_SIZE* to the affected window or control so it can adjust any settings needed to later paint its image in the new size. Your VCL component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM_SIZE* message, you will add a message-handling method to the component.

Creating a message-handling method is described in detail in the section Creating new message handlers.

In this case, the calendar control needs a response to *WM_SIZE*, so add a protected method called *WMSize* to the control indexed to the *WM_SIZE* message, then write the method so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  .
  .
  .
  end;
.
.
.
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;                                    { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                 { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7;     { set new default cell width }
```

```
  DefaultRowHeight := (Message.Height - GridLines) div 7;              { and cell height }
end;
```

```
[C++]

//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
protected:
    void __fastcall WMSize(TWMSize &Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize)
END_MESSAGE_MAP(TCustomGrid)
};
```

```
[C++]

//implementation file
void __fastcall TSampleCalendar::WMSize(TWMSize &Message)
{
  int GridLines;                                    // temporary local variable
  GridLines = 6 * GridLineWidth;                    // calculated combined size of all lines
  DefaultColWidth = (Message.Width - GridLines) / 7;    // set new default cell width
  DefaultRowHeight = (Message.Height - GridLines) / 7;  // and cell height
}
```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

In this case, the calendar control needs to override *BoundsChanged* so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  .
  .
  .
  end;
.
.
.
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                          { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                  { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7;              { and cell height }
  inherited; {now call the inherited method }
end;
```

```
[C++]

//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
protected:
    void __fastcall BoundsChanged(void);
};
```

```
[C++]

//implementation file
void __fastcall TSampleCalendar::BoundsChanged(void)
{
  int GridLines;                                  // temporary local variable
  GridLines = 6 * GridLineWidth;                  // calculated combined size of all lines
  DefaultColWidth = (Width - GridLines) / 7;    // set new default cell width
  DefaultRowHeight = (Height - GridLines) / 7;  // and cell height
  TCustomGrid::BoundsChanged(); // now call the inherited method
}
```

Filling in the Cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

```
[Delphi]

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
.
.
.
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL strings }
end;
```

```
[C++]


void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect &ARect,
```

```
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol + 1];
  else
  {
    TheText = "";
    TempDay = DayNum(ACol, ARow);                    // DayNum is defined later
    if (TempDay != -1) TheText = IntToStr(TempDay);
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left
   - Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

Tracking the Date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date
- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

Storing the Internal Date

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

1 Declare a private field to hold the date:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  .
  .
  .
```

```
[C++]
```

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
public:
    __property TDateTime CalendarDate = {read=FDate, write=SetCalendarDate, nodefault};
    .
    .
    .
};
```

```
[C++]

class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TDateTime FDate;
    .
    .
    .
};
```

2 Initialize the date field in the constructor:

```
[Delphi]
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);          { this is already here }
  .                                   { other initializations here }
  .
  .
  FDate := Date;                     { get current date from RTL }
end;
```

```
[C++]

__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  .
  .
  .
  FDate = FDate.CurrentDate();
}
```

3 Declare a runtime property to allow access to the encoded date.

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  .
  .
  .
```

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                { set new date value }
  Refresh;                       { update the onscreen image }
end;
```

```
[C++]

class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    void __fastcall SetCalendarDate(TDateTime Value);
    .
    .
    .
};
```

```
[C++]

void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                       // Set the new date value
  Refresh();                           // Update the onscreen image
}
```

Accessing the Day, Month, and Year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

1 Declare the three properties, assigning each a unique **index** number:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  .
  .
  .
```

```
[C++]
class PACKAGE TSampleCalendar : public TCustomGrid
```

```
{
  .
  .
  .
public:
    __property int Day = {read=GetDateElement, write=SetDateElement, index=3,
      nodefault};
    __property int Month = {read=GetDateElement, write=SetDateElement, index=2,
nodefault};
    __property int Year = {read=GetDateElement, write=SetDateElement, index=1,
nodefault};
};
```

2 Declare and write the implementation methods, setting different elements for each index value:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;         { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  .
  .
  .
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                      { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                 { get current date elements }
    case Index of                                    { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                { encode the modified date }
    Refresh;                                               { update the visible calendar }
  end;
end;
```

```
[C++]
// header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
```

```
    int __fastcall GetDateElement(int Index);           // note the Index parameter
    void __fastcall SetDateElement(int Index, int Value);
    .
    .
    .
};
```

```
[C++]
// implementation file
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);           // break encoded date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0)                                       // all elements must be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);     // get current date elements
    switch (Index)
    {
      case 1: AYear = Value;     break;
      case 2: AMonth = Value;    break;
      case 3: ADay = Value;      break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);         // encode the modified date
  Refresh();                                      // update the visible calendar
}
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

Generating the Day Numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class field, then update that field each time the date changes.

To fill in the days in the proper cells, you do the following:

1 Add a month-offset field to the object and a method that updates the field value:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                    { storage for the offset }
  .
  .
  .
  protected
    procedure UpdateCalendar; virtual;                    { property for offset access }
  end;
.
.
.
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                             { date of the first day of the month }
begin
  if FDate <> 0 then                              { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                     { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                     { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);        { generate the offset into the grid }
  end;
  Refresh;                                                  { always repaint the control }
end;
```

```
[C++]
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int FMonthOffset;                          // storage for the offset
    .
    .
    .
protected:
    virtual void __fastcall UpdateCalendar(void);
    .
    .
    .
};
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;                          // date of first day of the month
  if ((int)FDate != 0)                          // only calculate offset if date is valid
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);  // get elements of date
    FirstDate = TDateTime(AYear, AMonth, 1);   // date of the first
    FMonthOffset = 2 - FirstDate.DayOfWeek();  // generate the offset into the grid
  }
  Refresh();                                    // always repaint the control
}
```

2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```
[Delphi]
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                    { this is already here }
  .                                                          { other initializations here }
  .
  .
  UpdateCalendar;                                                { set proper offset }
end;
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin  FDate := Value;                                                { this was already here }
  UpdateCalendar;                                        { this previously called Refresh }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  .
  .
  .
    FDate := EncodeDate(AYear, AMonth, ADay);                 { encode the modified date }
    UpdateCalendar;                                      { this previously called Refresh }
  end;
end;
```

```
[C++]
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner)
  : TCustomGrid(Owner)
{
  .
  .
  .
  UpdateCalendar();
}
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                                    // this was already here
  UpdateCalendar();                                 // this previously called Refresh
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  .
  .
  .
  FDate = TDateTime(AYear, AMonth, ADay);         // this was already here
  UpdateCalendar();                               // this previously called Refresh
}
```

3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```
[Delphi]
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;          { calculate day for this cell }
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                                { return -1 if invalid }
end;
```

```
[C++]
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;       // calculate day for this cell
  if ((result < 1)||(result > MonthDays[IsLeapYear(Year)][Month]))
    result = -1;   // return -1 if invalid
  return result;
}
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4 Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```
[Delphi]
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                          { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                             { just use the day name }
  else begin
    TheText := '';                                            { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                             { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);         { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

```
[C++]
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0)                                    // this is the header row
    TheText = ShortDayNames[ACol + 1];              // just use the day name
  else
  {
    TheText = "";                                   // blank cell is the default
    TempDay = DayNum(ACol, ARow);                   // get number for this cell
    if (TempDay != -1) TheText = IntToStr(TempDay); // use the number if valid
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

Selecting the Current Day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
[Delphi]
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    .  { existing statements to set FMonthOffset }
    .
    .
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

```
[C++]
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int) FDate != 0)
  {
  .
  .
  .                                          // existing statements to set FMonthOffset
    Row = (ADay - FMonthOffset) / 7 + 1;
```

```
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();                                    // this is already here
}
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

Navigating Months and Years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a "next month" feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year.

```
[Delphi]
procedure TCalendar.NextMonth;
begin
  CalendarDate := IncMonth(CalendarDate, 1);
end;
procedure TCalendar.PrevMonth;
begin
  CalendarDate := IncMonth(CalendarDate, -1);
end;
procedure TCalendar.NextYear;
begin
  CalendarDate := IncMonth(CalendarDate, 12);
end;
procedure TCalendar.PrevYear;
begin
  CalendarDate := DecodeDate(IncMonth(CalendarDate, -12);
end;
```

```
[C++]
void __fastcall TSampleCalendar::NextMonth()
{
  CalendarDate = IncMonth(CalendarDate, 1);
}
void __fastcall TSampleCalendar::PrevMonth()
{
  CalendarDate = IncMonth(CalendarDate, -1);
}
void __fastcall TSampleCalendar::NextYear()
{
  CalendarDate = IncMonth(CalendarDate, 12);
}
void __fastcall TSampleCalendar::PrevYear()
{
  CalendarDate = IncMonth(CalendarDate, -12);
}
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

Navigating Days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Moving the Selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```
[Delphi]
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;                              { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);                 { get the day number for the clicked cell }
```

```
  if TempDay <> -1 then Day := TempDay;                              { change day if valid }
end;
```

```
[C++]
void __fastcall TSampleCalendar::Click()
{
  int TempDay = DayNum(Col, Row);            // get the day number for the clicked cell
  if (TempDay != -1) Day = TempDay;          // change day if valid
}
```

Providing an OnChange Event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an OnChange event to TSampleCalendar.

1 Declare the event, a field to store the event, and a dynamic method to call the event:

```
[Delphi]
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  .
  .
  .
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  .
  .
  .
```

```
[C++]
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TNotifyEvent FOnChange;
    .
    .
    .
protected:
    virtual void __fastcall Change();
__published:
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
    .
    .
    .
}
```

2 Write the *Change* method:

```
[Delphi]
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

```
[C++]
void __fastcall TSampleCalendar::Change()
{
  if(FOnChange != NULL) FOnChange(this);
}
```

3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```
[Delphi]
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                              { this is the only new statement }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    .                                             { many statements setting element values }
    .
    .
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                                { this is new }
  end;
end;
```

```
[C++]
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();              // this is the only new statement
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  .
  .
  .    // many statements setting element values
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();              // this is new
}
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

Excluding Blank Cells

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
[Delphi]
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False            { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
```

```
[C++]
bool __fastcall TSampleCalendar::SelectCell(int ACol, int ARow)
{
  if (DayNum(ACol,ARow) == -1) return false;          // -1 indicates invalid date
  else return TCustomGrid::SelectCell(ACol, ARow);    // otherwise, use inherited value
}
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

Making a control data aware

Making a Control Data Aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Using data controls .

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This section first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Customizing a grid You can also use the standard calendar control on the Samples page of the Tool palette, *TCalendar*.

The section then continues with an explanation of how to make the new data browsing control a data editing control.

Creating a Data Browsing Control

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component.
- Adding the data link.
- Responding to data changes.

Creating and registering the component

You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *DBCal*.
- Derive a new component class called *TDBCalendar*, descended from the component *TSampleCalendar*. The section Customizing a grid shows you how to create the *TSampleCalendar* component.

- Register *TDBCalendar* on the Samples page of the Tool palette.

The resulting unit descending from *TCustomGrid* in a VCL application should look like this:

```
unit CalSamp;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note: While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the DrawCell method is described in Filling in the cells.

Making the Control Read-only

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves:

- Adding the ReadOnly property.
- Allowing needed updates.

Note: Note that if you started with the *TCalendar* component from Delphi's Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

Adding the ReadOnly property

By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unable to be selected.

To add the ReadOnly property, follow these steps:

1 Add the property declaration and a **private** field to hold the value:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar)
  private
```

```
    FReadOnly: Boolean;                                    { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;     { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
.
.
.
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  FReadOnly := True;                                        { set the default value }
end;
```

```
[C++]

//header file
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    bool FReadOnly;                                  // field for internal storage
protected:
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
__published:
    __property ReadOnly = {read=FReadOnly, write=FReadOnly, default=true};
};
```

```
[C++]

//implementation file:
virtual __fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
  TSampleCalendar(Owner)
{
  FReadOnly = true;                                  // sets the default value
}
```

2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in Excluding blank cells.

```
[Delphi]
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                        { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
```

```
[C++]
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (FReadOnly) return false;                          // can't select if read only
  return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCalendar*, and append the **override** directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

Allowing Needed Updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                  { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;                { remember the override directive }
  end;
.
.
.
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False       { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;                                        { set flag to allow updates }
  try
    inherited UpdateCalendar;                                        { update as usual }
  finally
    FUpdating := False;                                    { always clear the flag }
  end;
end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    .
    .
    .
    bool FUpdating;                                  // private flag for internal use
protected:
    virtual bool __fastcall SelectCell(long ACol, long ARow);
public:
    .
    .
    .
    virtual void __fastcall UpdateCalendar();
```

```
    .
    .
    .
};
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
    if (!FUpdating && FReadOnly) return false;          // can't select if read only
    return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}
void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating=true;                                       // set flag to allow updates
  try
  {
     TSampleCalendar::UpdateCalendar();               // update as usual
  }
  catch(...)
  {
    FUpdating = false;
    throw;
  }
  FUpdating = false;                                    // always clear the flag
}
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data browsing ability.

Adding the Data Link

The connection between a control and a database is handled by a class called a *data link*. The data link class that connects a control with a single field in a database is *TFieldDataLink*. There are also data links for entire tables.

A data-aware control *owns* its data link class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Creating a graphic control

Establishing a data link as an owned class requires these three steps:

1. Declaring the class field.
2. Declaring the access properties.
3. Initializing the data link.

Declaring the Class Field

A component needs a field for each of its owned classes, as explained in Declaring the class fields. In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  .
  .
```

```
  .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    TFieldDataLink *FDataLink;
    .
    .
    .
};
```

```
[C++]
#include <DB.hpp>
#include <DBTables.hpp>
```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

Declaring the Access Properties for a Data-aware Control

Every data-aware control has a *DataSource* property that specifies which data source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Creating a graphic control these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the data link class.

Initializing the Data Link

A data-aware control needs access to its data link throughout its existence, so it must construct the data link object as part of its own constructor, and destroy the data link object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar)
  public                                  { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    .
    .
    .
  end;
.
.
.
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                 { always call the inherited constructor first }
```

```
  FDataLink := TFieldDataLink.Create;                    { construct the datalink object }
  FDataLink.Control := self;               {let the datalink know about the calendar }
  FReadOnly := True;                                              { this is already here }
end;
destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                                  { always destroy owned objects first... }
  inherited Destroy;                                  { ...then call inherited destructor }
end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
public:
    virtual __fastcall TDBCalendar(TComponent *Owner);
    __fastcall ~TDBCalendar();
};
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) : TSampleCalendar(Owner)
{
    FReadOnly = true;
    FDataLink = new TFieldDataLink();
    FDataLink->Control = this;
}
__fastcall TDBCalendar::~TDBCalendar()
{
    FDataLink->Control = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;
}
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

Responding to Data Changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Data link classes all have events named *OnDataChange*. When the data source indicates a change in its data, the data link object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

Creating a Data Editing Control

When you create a data editing control, you create and register the component and add the data link just as you do for a data browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data editing control described here is the same calendar control described in Creating a data browsing control. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data editing control involves:

- Changing the default value of FReadOnly.
- Handling mouse-down and key-down messages.
- Updating the field data link class.
- Modifying the Change method.
- Updating the dataset.

Changing the Default Value of FReadOnly

Because this is a data editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:

```
[Delphi]
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  .
  .
  .
  FReadOnly := False;  { set the default value }
  .
  .
  .
end;
```

```
[C++]
__fastcall TDBCalendar::TDBCalendar (TComponent* Owner) : TSampleCalendar(Owner)
{
  FReadOnly = false;             // set the default value
  .
  .
  .
}
```

Handling Mouse-down and Key-down Messages

When the user of the control begins interacting with it, the control receives either mouse-down messages (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN*, or *WM_RBUTTONDOWN*) or a key-down message (*WM_KEYDOWN*) from Windows. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down messages.
- Responding to key-down messages.

Responding to Mouse-down Messages

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a Windows mouse-down message. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
   .
   .
   .
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   .
   .
   .
  end;
procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

```
[C++]

//header file
class PACKAGE TDBCalendar : public TSampleCalendar
{
.
.
.
protected:
    virtual void __fastcall MouseDown(TMouseButton Button,  TShiftState Shift, int X,
int Y);
    .
    .
    .
};
```

```
[C++]

//implmentation file
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,  TShiftState Shift, int  X,
int Y)
{
  TMouseEvent MyMouseDown;                              // declare event type
  if (!FReadOnly && FDataLink->Edit())                  // if the field can be edited
    TSampleCalendar::MouseDown(Button, Shift, X, Y);   // call the inherited MouseDown
```

```
  else
  {
    MyMouseDown = OnMouseDown;                          // assign OnMouseDown event
    if (MyMouseDown != NULL) MyMouseDown(this, Button,  // execute code in the...
       Shift, X, Y);                                    // ...OnMouseDown event handler
  }
}
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

Responding to Key-down Messages

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override KeyDown, follow these steps:

1 Add a *KeyDown* method to the *TDBCalendar* class:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
    .
    .
    .
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
    .
    .
    .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
    .
    .
    .
protected:
    virtual void __fastcall KeyDown(unsigned short &Key,  TShiftState Shift);
    .
    .
    .
};
```

2 Implement the *KeyDown* method:

```
[Delphi]
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
```

```
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

```
[C++]

void __fastcall TDBCalendar::KeyDown(unsigned short &Key,  TShiftState Shift)
{
  TKeyEvent MyKeyDown;                                  // declare event type
  Set<unsigned short,0,8> keySet;
  keySet = keySet << VK_UP << VK_DOWN << VK_LEFT      // assign virtual keys to set
    << VK_RIGHT << VK_END << VK_HOME << VK_PRIOR << VK_NEXT;
  if (!FReadOnly &&                                 // if control is not read only...
      (keySet.Contains(Key)) &&                      // ...and key is in the set...
      FDataLink->Edit() )                            // ...and field is in edit mode
  {
    TCustomGrid::KeyDown(Key, Shift);          // call the inherited KeyDown method
  }
  else
  {
    MyKeyDown = OnKeyDown;                              // assign OnKeyDown event
    if (MyKeyDown != NULL) MyKeyDown(this,Key,Shift); // execute code in...
  }                                                        // ...OnKeyDown event handler
}
```

When *KeyDown* responds to a mouse-down message, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

Updating the Field Data Link Class

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field data link class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

To reflect a change made to the value in the calendar in the field value:

1 Add an *UpdateData* method to the private section of the calendar component:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
   .
   .
   .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall UpdateData(TObject *Sender);
};
```

2 Implement the *UpdateData* method:

```
[Delphi]
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;        { set field link to calendar date }
end;
```

```
[C++]
void __fastcall TDBCalendar::UpdateData( TObject* Sender)
{
    FDataLink->Field->AsDateTime = CalendarDate;     // set field link to calendar date
}
```

3 Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
[Delphi]
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

```
[C++]
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
  : TSampleCalendar(Owner)
{
  FDataLink = new TFieldDataLink();       // this was already here
  FDataLink->OnDataChange = DataChange;   // this was here too
  FDataLink->OnUpdateData = UpdateData;   // assign UpdateData to the OnUpdateData event
}
```

Modifying the Change Method

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code.

These are the steps to follow:

1 Add a new *Change* method to the *TDBCalendar* component:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   .
   .
   .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
protected:
    virtual void __fastcall Change();
    .
    .
    .
};
```

2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
[Delphi]
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                  { call the Modified method }
  inherited Change;                    { call the inherited Change method }
end;
```

```
[C++]
void __fastcall TDBCalendar::Change()
{
  if (FDataLink != NULL)
    FDataLink->Modified();                // call the Modified method
  TSampleCalendar::Change();              // call the inherited Change method
}
```

Updating the Dataset

So far, a change within the data-aware control has changed values in the field data link class. The final step in creating a data editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the Tab key.

Note: VCL applications define message control IDs for operations on controls. For example, the *CM_EXIT* message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this case, when the user exits the control, the *CMExit* method, the message handler for *CM_EXIT*, responds by updating the record in the dataset with the changed values in the field data link class. For more information about message handlers, see Handling messages and system notifications.

To update the dataset within a message handler, follow these steps:

1 Add the message handler to the *TDBCalendar* component:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
   .
   .
   .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall CMExit(TWMNoParams Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP
};
```

2 Implement the *CMExit* method so it looks like this:

```
[Delphi]
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                        { tell data link to update database }
```

```
  except
    on Exception do SetFocus;                    { if it failed, don't let focus leave }
 end;
  inherited;
end;
```

```
[C++]
void __fastcall TDBCalendar::CMExit(TWMNoParams &Message)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don't let focus leave
    throw;
  }
}
```

To update the dataset when the user exits the control, follow these steps:

1 Add an override for the *DoExit* method to the *TDBCalendar* component:

```
[Delphi]
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
   .
   .
   .
  end;
```

```
[C++]
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    DYNAMIC void __fastcall DoExit(void);
   .
   .
   .
};
```

2 Implement the *DoExit* method so it looks like this:

```
[Delphi]
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                    { tell data link to update database }
  except
    on Exception do SetFocus;                  { if it failed, don't let focus leave }
 end;
```

```
  inherited;                          { let the inherited method generate an OnExit event }
end;
```

```
[C++]
void __fastcall TDBCalendar::DoExit(void)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don't let focus leave
    throw;
  }
  TCustomGrid::DoExit(); // let the inherited method generate an OnExit event
}
```

Making a dialog box a component

Making a Dialog Box a Component: Overview

You will find it convenient to make a frequently used dialog box into a component that you add to the Tool palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

1. Defining the component interface
2. Creating and registering the component
3. Creating the component interface
4. Testing the component

The Delphi "wrapper" component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this section, you will see how to create a wrapper component around the generic About Box form provided in the Delphi Object Repository.

Note: Copy the files ABOUT.PAS and ABOUT.DFM into your working directory.

There are not many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

Defining the Component Interface

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

Creating and Registering the Component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *AboutDlg*.
- Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
- Register *TAboutBoxDlg* on the Samples page of the Tool palette.

The resulting unit should look like this:

```
[Delphi]
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

```
[C++]
#include <vcl\vcl.h>
#pragma hdrstop
#include "AboutDlg.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TAboutBoxDlg *ValidCtrCheck()
{
  return new TAboutBoxDlg(NULL);
}
//---------------------------------------------------------------------------
namespace AboutDlg {
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

```
[C++]
#ifndef AboutDlgH
```

```
#define AboutDlgH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section, you will create the interface between the component and the dialog box.

Creating the Component Interface

These are the steps to create the component interface:

1. Including the form unit files.
2. Adding interface properties.
3. Adding the Execute method.

Including the Form Unit

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit.

Append *About* to the **uses** clause of the *AboutDlg* unit.

The **uses** clause now looks like this:

```
[Delphi]
uses
  Windows, SysUtils, Messages, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms,
  About;
```

```
[C++]
// for C++
#include "About.h"
#pragma link "About.obj"
```

The form unit always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the *About* unit includes the following declaration:

```
[Delphi]
var
  AboutBox: TAboutBox;
```

```
[C++]
extern TAboutBox *AboutBox;
```

So by adding *About* to the **uses** clause, you make *AboutBox* available to the wrapper component.

Adding Interface Properties

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private class field, which is a variable the wrapper uses to store the value of the property
- The published property declaration itself, which specifies the name of the property and tells it which field to use for storage

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the class field that stores the property's value has the same name as the property, but with the letter *F* in front. The field and the property *must* be of the same type.

Adding the Execute Method

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common dialog box components, you use a boolean function called *Execute* that returns *True* if the user clicks OK, or *False* if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

```
[Delphi]
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

```
[C++]
class PACKAGE TMyWrapper : public TComponent
{
  .
  .
  .
public:
    bool __fastcall Execute();
  .
```

```
  .
  .
};
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either *True* or *False*, depending on the return value from *ShowModal*.

Testing the Component

Once you have installed the dialog box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Tool palette, you can test it with the following steps:

1. Create a new project.
2. Place an About box component on the main form.
3. Place a command button on the form.
4. Double-click the command button to create an empty click-event handler.
5. In the click-event handler, type the following line of code:

```
[Delphi]
AboutBoxDlg1.Execute;
```

```
[C++]
AboutBoxDlg1->Execute();
```

6. Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About box component and again running the application.

Extending the IDE

Extending the IDE

You can extend and customize the IDE with your own menu items, tool bar buttons, dynamic form-creation wizards, and more, using the Open Tools API (often shortened to just Tools API). The Tools API is a suite of over 100 interfaces that interact with and control the IDE, including the main menu, the tool bars, the main action list and image list, the source editor's internal buffers, keyboard macros and bindings, forms and their components in the form editor, the debugger and the process being debugged, code completion, the message view, and the To-Do list.

Using the Tools API is simply a matter of writing classes that implement certain interfaces, and calling on services provided by other interfaces. Your Tools API code must be compiled and loaded into the IDE at design-time as a design-time package or in a DLL. Thus, writing a Tools API extension is somewhat like writing a property or component editor. Before tackling this material, make sure you are familiar with the basics of working with packages and registering components.

The following topics describe how to use the Tools API:

- Overview of the Tools API
- Writing a wizard class
- Obtaining Tools API services
- Working with files and editors
- Creating forms and projects
- Notifying a wizard of IDE events

Overview of the Tools API

All of the Tools API declarations reside in a single unit, ToolsAPI. To use the Tools API, you typically use the designide package, which means you must build your Tools API add-in as a design-time package or as a DLL that uses runtime packages. For information about package and library issues, see Installing the wizard package.

The main interface for writing a Tools API extension is *IOTAWizard*, so most IDE add-ins are called wizards. C++Builder and Delphi wizards are, for the most part, interoperable. You can write and compile a wizard in Delphi, then use it in C++Builder, and vice versa. Interoperability works best with the same version number, but it is also possible to write wizards so they can be used in future versions of both products.

To use the Tools API, you write wizard classes that implement one or more of the interfaces defined in the ToolsAPI unit.

A wizard makes use of services that the Tools API provides. Each service is an interface that presents a set of related functions. The implementation of the interface is hidden within the IDE. The Tools API publishes only the interface,

which you can use to write your wizards without concerning yourself with the implementation of the interfaces. The various services offer access to the source editor, form designer, debugger, and so on. See Obtaining Tools API services for information about using the interfaces that expose services to your wizard.

The service and other interfaces fall into two basic categories. You can tell them apart by the prefix used for the type name:

- The NTA (native tools API) grants direct access to actual IDE objects, such as the IDE's *TMainMenu* object. When using these interfaces, the wizard must use Borland packages, which also means the wizard is tied to a specific version of the IDE. The wizard can reside in a design-time package or in a DLL that uses runtime packages.
- The OTA (open tools API) does not require packages and accesses the IDE only through interfaces. In theory, you could write a wizard in any language that supports COM-style interfaces, provided you can also work with the Delphi calling conventions and Delphi types such as AnsiString. OTA interfaces do not grant full access to the IDE, but almost all the functionality of the Tools API is available through OTA interfaces. If a wizard uses only OTA interfaces, it is possible to write a DLL that is not dependent on a specific version of the IDE.

The Tools API has two kinds of interfaces: those that you, the programmer, must implement and those that the IDE implements. Most of the interfaces are in the latter category: the interfaces define the capability of the IDE but hide the actual implementation. The kinds of interfaces that you must implement fall into three categories: wizards, notifiers, and creators:

- As mentioned earlier in this topic, a wizard class implements the *IOTAWizard* interface and possibly derived interfaces.
- A notifier is another kind of interface in the Tools API. The IDE uses notifiers to call back to your wizard when something interesting happens. You write a class that implements the notifier interface, register the notifier with the Tools API, and the IDE calls back to your notifier object when the user opens a file, edits source code, modifies a form, starts a debugging session, and so on. Notifiers are covered in Notifying a wizard of IDE events .
- A creator is another kind of interface that you must implement. The Tools API uses creators to create new units, projects, or other files, or to open existing files. See Creating forms and projects for information about creator interfaces.

Other important interfaces are modules and editors. A module interface represents an open unit, which has one or more files. An editor interface represents an open file. Different kinds of editor interfaces give you access to different aspects of the IDE: the source editor for source files, the form designer for form files, and project resources for a resource file. See Working with files and editors for information about module and editor interfaces.

Writing a Wizard Class

There are four kinds of wizards, where the wizard kind depends on the interfaces that the wizard class implements. The following table describes the four kinds of wizards.

The four kinds of wizards

Interface	Description
IOTAFormWizard	Typically creates a new unit, form, or other file
IOTAMenuWizard	Automatically added to Help menu
IOTAProjectWizard	Typically creates a new application or other project
IOTAWizard	Miscellaneous wizard that doesn't fit into other categories

The four kinds of wizards differ only in how the user invokes the wizard:

- A menu wizard is added to the IDE's Help menu. When the user picks the menu item, the IDE calls the wizard's *Execute* function. Plain wizards offer much more flexibility, so menu wizards are typically used only for prototypes and debugging.

- Form and project wizards are called repository wizards because they reside in the Object Repository. The user invokes these wizards from the New Items dialog box. The user can also see the wizards in the object repository (by choosing the **Tools** ▸ **Repository** menu item). The user can check the New Form check box for a form wizard, which tells the IDE to invoke the form wizard when the user chooses the **File** ▸ **New** ▸ **Form** menu item. The user can also check the Main Form check box. This tells the IDE to use the form wizard as the default form for a new application. The user can check the New Project check box for a project wizard. When the user chooses **File** ▸ **New** ▸ **Application**, the IDE invokes the selected project wizard.
- The fourth kind of wizard is for situations that don't fit into the other categories. A plain wizard does not do anything automatically or by itself. Instead, you must define how the wizard is invoked.

The Tools API does not enforce any restrictions on wizards, such as requiring a project wizard to create a project. You can just as easily write a project wizard to create a form and a form wizard to create a project (if that's something you really want to do).

The following topics provide details on how to implement and install a wizard:

- Implementing the wizard interfaces
- Installing the wizard package

Implementing the Wizard Interfaces

Every wizard class must implement at least *IOTAWizard*, which requires implementing its ancestors, too: *IOTANotifier* and *IInterface*. Form and project wizards must implement all their ancestor interfaces, namely, *IOTARepositoryWizard*, *IOTAWizard*, *IOTANotifier*, and *IInterface*.

For C++, to use NotifierObject as a base class you must use multiple inheritance. Your wizard class must inherit from NotifierObject and from the wizard interfaces that you need to implement, such as IOTAWizard. Because IOTAWizard inherits from IOTANotifier and IInterface, there is an ambiguity in the derived class: functions such as AddRef() are declared in every branch of the ancestral inheritance graph. To resolve this problem, pick one base class as the primary base class and delegate all ambiguous functions to that one class. For example, the class declaration might look as follows:

```
class PACKAGE MyWizard : public NotifierObject, public IOTAMenuWizard {
    typedef NotifierObject inherited;
    public:
        // IOTAWizard
        virtual AnsiString __fastcall GetIDString();
        virtual AnsiString __fastcall GetName();
        virtual TWizardState __fastcall GetState();
        virtual void __fastcall Execute();
        // IOTAMenuWizard
        virtual AnsiString __fastcall GetMenuText();
      void __fastcall AfterSave();
        void __fastcall BeforeSave();
        void __fastcall Destroyed();
        void __fastcall Modified();
    protected:
        // IInterface
        virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
};
```

```
// implementation
ULONG __stdcall MyWizard::AddRef()  { return inherited::AddRef(); }
```

```
ULONG __stdcall MyWizard::Release() { return inherited::Release(); }
HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void** obj)
{
    if (iid == __uuidof(IOTAMenuWizard)) {
        *obj = static_cast<IOTAMenuWizard*>(this);
        static_cast<IOTAMenuWizard*>(*obj)->AddRef();
        return S_OK;
    }
    if (iid == __uuidof(IOTAWizard)) {
        *obj = static_cast<IOTAWizard*>(this);
        static_cast<IOTAWizard*>(*obj)->AddRef();
        return S_OK;
    }
    return inherited::QueryInterface(iid, obj);
}
```

Your implementation of *IInterface* must follow the normal rules for Delphi interfaces, which are the same as the rules for COM interfaces. That is, *QueryInterface* performs type casts, and *_AddRef* and *_Release* manage reference counting. You might want to use a common base class to simplify writing wizard and notifier classes. For this purpose, the ToolsAPI unit defines a class, *TNotifierObject*, which implements *IOTANotifier* interface with empty method bodies.

You can write a class similar to *TNotifierObject* in C++.

```
class PACKAGE NotifierObject : public IOTANotifier {
    public:
        __fastcall NotifierObject() : ref_count(0) {}
        virtual __fastcall ~NotifierObject();
        void __fastcall AfterSave();
        void __fastcall BeforeSave();
        void __fastcall Destroyed();
        void __fastcall Modified();
    protected:
        // IInterface
        virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
    private:
        long ref_count;
};
```

```
// implementation
ULONG __stdcall NotifierObject::AddRef()
{
    return InterlockedIncrement(&ref_count);
}
ULONG __stdcall NotifierObject::Release()
{
    ULONG result = InterlockedDecrement(&ref_count);
    if (ref_count == 0)
        delete this;
    return result;
}
HRESULT __stdcall NotifierObject::QueryInterface(const GUID& iid, void** obj)
{
    if (iid == __uuidof(IInterface)) {
        *obj = static_cast<IInterface*>(this);
        static_cast<IInterface*>(*obj)->AddRef();
```

```
        return S_OK;
    }
    if (iid == __uuidof(IOTANotifier)) {
        *obj = static_cast<IOTANotifier*>(this);
        static_cast<IOTANotifier*>(*obj)->AddRef();
        return S_OK;
    }
    return E_NOINTERFACE;
}
```

Although wizards inherit from *IOTANotifier*, and must therefore implement all of its functions, the IDE does not usually make use of those functions, so your implementations can be empty (as they are in *TNotifierObject*). Thus, when you write your wizard class, you need only declare and implement those interface methods introduced by the wizard interfaces, accepting the *TNotifierObject* implementation of *IOTANotifier*.

```
// C++ empty implementations
void __fastcall NotifierObject::AfterSave()  {}
void __fastcall NotifierObject::BeforeSave() {}
void __fastcall NotifierObject::Destroyed()  {}
void __fastcall NotifierObject::Modified()   {}
```

Installing the Wizard Package

As with any other design-time package, a wizard package must have a *Register* function. (See Registering components for details about the *Register* function.) In the *Register* function, you can register any number of wizards by calling *RegisterPackageWizard*, and passing a wizard object as the sole argument, as shown below:

```
[Delphi]
procedure Register;
begin
  RegisterPackageWizard(MyWizard.Create);
  RegisterPackageWizard(MyOtherWizard.Create);
end;
```

```
[C++]
namespace Example {
    void __fastcall PACKAGE Register()
    {
        RegisterPackageWizard(new MyWizard());
        RegisterPackageWizard(new MyOtherWizard());
    }
}
```

You can also register property editors, components, and so on, as part of the same package.

Remember that a design-time package is part of the main Developer Studio 2006 application, which means any form names must be unique throughout the entire application and all other design-time packages. This is the main disadvantage to using packages: you never know what someone else might name their forms.

During development, install the wizard package the way you would any other design-time package: click the Install button in the package manager. The IDE will compile and link the package and attempt to load it. The IDE displays a dialog box telling you whether it successfully loaded the package.

Obtaining Tools API Services

To do anything useful, a wizard needs access to the IDE: its editors, windows, menus, and so on. This is the role of the service interfaces. The Tools API includes many services, such as action services to perform file actions, editor services to access the source code editor, debugger services to access the debugger, and so on. The following table summarizes all the service interfaces.

Tools API service interfaces

Interface	Description
INTAServices	Provides access to native IDE objects: main menu, action list, image list, and tool bars.
IOTAActionServices	Performs basic file actions: open, close, save, and reload a file.
IOTACodeCompletionServices	Provides access to code completion, allowing a wizard to install a custom code completion manager.
IOTADebuggerServices	Provides access to debugger.
IOTAEditorServices	Provides access to source code editor and its internal buffers.
IOTAKeyBindingServices	Permits a wizard to register custom keyboard bindings.
IOTAKeyboardServices	Provides access to keyboard macros and bindings.
IOTAKeyboardDiagnostics	Toggle debugging of keystrokes.
IOTAMessageServices	Provides access to message view.
IOTAModuleServices	Provides access to open files.
IOTAPackageServices	Queries the names of all installed packages and their components.
IOTAServices	Miscellaneous services.
IOTAToDoServices	Provides access to the To-Do list, allowing a wizard to install a custom To-Do manager.
IOTAToolsFilter	Registers tools filter notifiers.
IOTAWizardServices	Registers and unregisters wizards.

To use a service interface, cast the *BorlandIDEServices* variable to the desired service using the global Supports function, which is defined in the SysUtils unit. For example,

```
[Delphi]
procedure set_keystroke_debugging(debugging: Boolean);
var
  diag: IOTAKeyboardDiagnostics
begin
  if Supports(BorlandIDEServices, IOTAKeyboardDiagnostics, diag) then
    diag.KeyTracing := debugging;
end;
```

```
[C++]
void set_keystroke_debugging(bool debugging)
{
_di_IOTAKeyboardDiagnostics diag;
if (BorlandIDEServices->Supports(diag))
diag->KeyTracing = debugging;
}
```

If your wizard needs to use a specific service often, you can keep a pointer to the service as a data member of your wizard class.

The following topics discuss special considerations when working with the Tools API service interfaces:

- Using native IDE objects
- Debugging a wizard
- Interface version numbers

Using Native IDE Objects

Wizards have full access to the main menu, tool bars, action list, and image list of the IDE. (Note that the IDE's many context menus are not accessible through the Tools API.)

The starting point for working with native IDE objects is the *INTAServices* interface. Use this interface to add an image to the image list, an action to the action list, a menu item to the main menu, and a button to a tool bar. You can tie the action to the menu item and tool button. When the wizard is destroyed, it must clean up the objects it creates, but it must not delete the image it added to the image list. Deleting an image would scramble the indices for all images added after this wizard.

The wizard uses the actual *TMainMenu*, *TActionList*, *TImageList*, and *TToolBar* objects from the IDE, so you can write code the way you would any other application. It also means you have a lot of scope for crashing the IDE or otherwise disabling important features, such as deleting the **File** menu. Debugging a wizard discusses steps you can take to debug your wizard if you find it has caused problems like these.

The following topics illustrate how to perform these tasks:

- Adding an image to the image list
- Adding an action to the action list
- Deleting toolbar buttons

Adding an Image to the Image List

Suppose you want to add a menu item to invoke your wizard. You also want to enable the user to add a toolbar button that invokes the wizard. The first step is to add an image to the IDE's image list. The index of your image can then be used for the action, which in turn is used by the menu item and toolbar button. Create a resource file that contains a 16 by 16 bitmap resource. Add the following code to your wizard's constructor:

```
[Delphi]
constructor MyWizard.Create;
var
  Services: INTAServices;
  Bmp: TBitmap;
  ImageIndex: Integer;
begin
  inherited;
  Supports(BorlandIDEServices, INTAServices, Services);
  { Add an image to the image list. }
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(HInstance, 'Bitmap1');
  ImageIndex := Services.AddMasked(Bmp, Bmp.TransparentColor,
                                   'Tempest Software.intro wizard image');
  Bmp.Free;
end;
```

```
[C++]
_di_INTAServices services;
BorlandIDEServices->Supports(services);
```

```
// Add an image to the image list.
Graphics::TBitmap* bitmap(new Graphics::TBitmap());
bitmap->LoadFromResourceName(reinterpret_cast<unsigned>(HInstance), "Bitmap1");
int image = services->AddMasked(bitmap, bitmap->TransparentColor,
                                 "Tempest Software.intro wizard image");
delete bitmap;
```

Be sure to load the resource by the name or ID you specify in the resource file. You must choose a color that will be interpreted as the background color for the image. If you don't want a background color, choose a color that does not exist in the bitmap.

Adding an Action to the Action List

The image index obtained in Adding an image to the image list is used to create an action, as shown below. The wizard uses the *OnExecute* and *OnUpdate* events. A common scenario is for a wizard to use the *OnUpdate* event to enable or disable the action. Be sure the *OnUpdate* event returns quickly, or the user will notice that the IDE becomes sluggish after loading your wizard. The action's *OnExecute* event is similar to the wizard's *Execute* method. If you are using a menu item to invoke a form or project wizard, you might even want to have *OnExecute* call *Execute* directly.

```
[Delphi]
NewAction := TAction.Create(nil);
NewAction.ActionList := Services.ActionList;
NewAction.Caption    := GetMenuText();
NewAction.Hint       := 'Display a silly dialog box';
NewAction.ImageIndex := ImageIndex;
NewAction.OnUpdate   := action_update;
NewAction.OnExecute  := action_execute;
```

```
[C++]
action = new TAction(0);
action->ActionList = services->ActionList;
action->Caption    = GetMenuText();
action->Hint       = "Display a silly dialog box";
action->ImageIndex = image;
action->OnUpdate   = action_update;
action->OnExecute  = action_execute;
```

The menu item sets its *Action* property to the newly created action. The tricky part of creating the menu item is knowing where to insert it. The example below looks for the**View** menu, and inserts the new menu item as the first item in the **View** menu. (In general, relying on absolute position is not a good idea: you never know when another wizard might insert itself in the menu. Future versions of Delphi are likely to reorder the menu, too. A better approach is to search the menu for a menu item with a specific name. The simplistic approach follows for the sake of clarity.)

```
[Delphi]
for I := 0 to Services.MainMenu.Items.Count - 1 do
begin
  with Services.MainMenu.Items[I] do
  begin
    if CompareText(Name, 'ViewsMenu') = 0 then
    begin
      NewItem := TMenuItem.Create(nil);
      NewItem.Action := NewAction;
      Insert(0, NewItem);
    end;
```

```
  end;
end;
```

```
[C++]
for (int i = 0; i < services->MainMenu->Items->Count; ++i)
{
TMenuItem* item = services->MainMenu->Items->Items[i];
if (CompareText(item->Name, "ViewsMenu") == 0)
{
menu_item = new TMenuItem(0);
menu_item->Action = action;
item->Insert(0, menu_item);
}
}
```

By adding the action to the IDE's action list, the user can see the action when customizing the toolbars. The user can select the action and add it as a button to any toolbar. This causes a problem when your wizard is unloaded: all the tool buttons end up with dangling pointers to the non-existent action and *OnClick* event handler. To prevent access violations, your wizard must find all tool buttons that refer to its action, and remove those buttons.

Deleting Toolbar Buttons

There is no convenient function for removing a button from a toolbar; you must send the CM_CONTROLCHANGE message, where the first parameter is the control to change, and the second parameter is zero to remove it or non-zero to add it to the toolbar. After removing the toolbar buttons, the destructor deletes the action and menu item. Deleting these items automatically removes them from the IDE's ActionList and MainMenu.

```
[Delphi]
procedure remove_action (Action: TAction; ToolBar: TToolBar);
var
  I: Integer;
  Btn: TToolButton;
begin
  for I := ToolBar.ButtonCount - 1 downto 0 do
  begin
    Btn := ToolBar.Buttons[I];
    if Btn.Action = Action then
    begin
      { Remove "Btn" from "ToolBar" }
      ToolBar.Perform(CM_CONTROLCHANGE, WPARAM(Btn), 0);
      Btn.Free;
    end;
  end;
end;
destructor MyWizard.Destroy;
var
  Services: INTAServices;
  Btn: TToolButton;
begin
  Supports(BorlandIDEServices, INTAServices, Services);
  { Check all the toolbars, and remove any buttons that use this action. }
remove_action(NewAction, Services.ToolBar[sCustomToolBar]);
remove_action(NewAction, Services.ToolBar[sDesktopToolBar]);
remove_action(NewAction, Services.ToolBar[sStandardToolBar]);
remove_action(NewAction, Services.ToolBar[sDebugToolBar]);
remove_action(NewAction, Services.ToolBar[sViewToolBar]);
remove_action(NewAction, Services.ToolBar[sInternetToolBar]);
```

```
  NewItem.Free;
  NewAction.Free;
end;
```

```
[C++]
void __fastcall remove_action (TAction* action, TToolBar* toolbar)
{
for (int i = toolbar->ButtonCount; --i >= 0; )
{
TToolButton* button = toolbar->Buttons[i];
if (button->Action == action)
{
// Remove "button" from "toolbar".
toolbar->Perform(CM_CONTROLCHANGE, WPARAM(button), 0);
delete button;
}
}
}
__fastcall MyWizard::~MyWizard()
{
_di_INTAServices services;
  BorlandIDEServices->Supports(services);
  // Check all the toolbars, and remove any buttons that use
// this action.
remove_action(action, services->ToolBar[sCustomToolBar]);
remove_action(action, services->ToolBar[sDesktopToolBar]);
remove_action(action, services->ToolBar[sStandardToolBar]);
remove_action(action, services->ToolBar[sDebugToolBar]);
remove_action(action, services->ToolBar[sViewToolBar]);
remove_action(action, services->ToolBar[sInternetToolBar]);
  delete menu_item;
delete action;
}
```

Debugging a Wizard

The Tools API provides you with a lot of flexibility in how your wizard interacts with the IDE. With the flexibility comes responsibility, however. It is easy to wind up with dangling pointers or other access violations.

When writing wizards that use the native tools API, you can write code that causes the IDE to crash. It is also possible that you write a wizard that installs but does not act the way you want it to. One of the challenges of working with design-time code is debugging. It's an easy problem to solve, however. Because the wizard is installed in Delphi itself, you simply need to set the package's Host Application to the Delphi executable from the **Run ▸ Parameters**... menu item.

When you want (or need) to debug the package, don't install it. Instead, choose **Run ▸ Run** from the menu bar. This starts up a new instance of Delphi. In the new instance, install the already-compiled package by choosing **Components ▸ Install Package**... from the menu bar. Back in the original instance of Delphi, you should now see the telltale blue dots that tell you where you can set breakpoints in the wizard source code. (If not, double-check your compiler options to be sure you enabled debugging; make sure you loaded the right package; and double-check the process modules to make extra sure that you loaded the .bpl file you wanted to load.)

You cannot debug into the VCL or RTL code this way, but you have full debug capabilities for the wizard itself, which might be enough to tell what is going wrong.

Interface Version Numbers

If you look closely at the declarations of some of the interfaces, such as *IOTAMessageServices*, you will see that they inherit from other interfaces with similar names, such as *IOTAMessageServices50*, which inherits from *IOTAMessageServices40*. This use of version numbers helps insulate your code from changes between releases of Delphi.

The Tools API follows the basic principle of COM, namely, that an interface and its GUID never change. If a new release adds features to an interface, the Tools API declares a new interface that inherits from the old one. The GUID remains the same, attached to the old, unchanged interface. The new interface gets a brand new GUID. Old wizards that use the old GUIDs continue to work.

The Tools API also changes interface names to try to preserve source-code compatibility. To see how this works, it is important to distinguish between the two kinds of interfaces in the Tools API: Borland-implemented and user-implemented. If the IDE implements the interface, the name stays with the most recent version of the interface. The new functionality does not affect existing code. The old interfaces have the old version number appended.

For a user-implemented interface, however, new member functions in the base interface require new functions in your code. Therefore, the name tends to stick with the old interface, and the new interface has a version number tacked onto the end.

For example, consider the message services. Delphi 6 introduced a new feature: message groups. Therefore, the basic message services interface required new member functions. These functions were declared in a new interface class, which retained the name *IOTAMessageServices*. The old message services interface was renamed to *IOTAMessageServices50* (for version 5). The GUID of the old *IOTAMessageServices* is the same as the GUID of the new *IOTAMessageServices50* because the member functions are the same.

Consider
IOTAIDENotifier as an example of a user-implemented interface. Delphi 5 added new overloaded functions: *AfterCompile* and *BeforeCompile*. Existing code that used *IOTAIDENotifier* did not need to change, but new code that required the new functionality had to be modified to override the new functions inherited from *IOTAIDENotifier50*. Version 6 did not add any more functions, so the current version to use is *IOTAIDENotifier50*.

The rule of thumb is to use the most-derived class when writing new code. Leave the source code alone if you are merely recompiling an existing wizard under a new release of Delphi.

Working with Files and Editors

It is important to understand how the Tools API works with files. The main interface is *IOTAModule*. A module represents a set of logically related open files. For example, a single module represents a single unit. The module, in turn, has one or more editors, where each editor represents one file, such as the unit source (.pas) or form (.dfm or .xfm) file. The editor interfaces reflect the internal state of the IDE's editors, so a wizard can see the modified code and forms that the user sees, even if the user has not saved any changes.

The following topics provide information about the module and editor interfaces:

- Using module interfaces
- Using editor interfaces

Using Module Interfaces

To obtain a module interface, start with the module services (*IOTAModuleServices*). You can query the module services for all open modules, look up a module from a file name or form name, or open a file to obtain its module interface.

There are different kinds of modules for different kinds of files, such as projects, resources, and type libraries. Cast a module interface to a specific kind of module interface to learn whether the module is of that type. For example, one way to obtain the current project group interface is as follows:

```
[Delphi]
{ Return the current project group, or nil if there is no project group. }
function CurrentProjectGroup: IOTAProjectGroup;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
begin
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProjectGroup, Result) then
      Exit;
  end;
  Result := nil;
end;
```

```
[C++]
// Return the current project group, or 0 if there is no project group.
_di_IOTAProjectGroup __fastcall CurrentProjectGroup()
{
_di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
  for (int i = 0; i < svc->ModuleCount; ++i)
{
_di_IOTAModule module = svc->Modules[i];
_di_IOTAProjectGroup group;
if (module->Supports(group))
return group;
}
return 0;
}
```

Using Editor Interfaces

Every module has at least one editor interface. Some modules have several editors, such as a source (.pas) file and form description (.dfm) file. All editors implement the *IOTAEditor* interface; cast the editor to a specific type to learn what kind of editor it is. For example, to obtain the form editor interface for a unit, you can do the following:

```
[Delphi]
{ Return the form editor for a module, or nil if the unit has no form. }
function GetFormEditor(Module: IOTAModule): IOTAFormEditor;
var
  I: Integer;
  Editor: IOTAEditor;
begin
  for I := 0 to Module.ModuleFileCount - 1 do
  begin
    Editor := Module.ModuleFileEditors[I];
    if Supports(Editor, IOTAFormEditor, Result) then
      Exit;
```

```
  end;
  Result := nil;
end;
```

```
[C++]
// Return the form editor for a module, or 0 if the unit has no form.
_di_IOTAFormEditor __fastcall GetFormEditor(_di_IOTAModule module)
{
for (int i = 0; i < module->ModuleFileCount; ++i)
{
_di_IOTAEditor editor = module->ModuleFileEditors[i];
_di_IOTAFormEditor formEditor;
if (editor->Supports(formEditor))
return formEditor;
}
return 0;
}
```

The editor interfaces give you access to the editor's internal state. You can examine the source code or components that the user is editing, make changes to the source code, components, or properties, change the selection in the source and form editors, and carry out almost any editor action that the end user can perform.

Using a form editor interface, a wizard can access all the components on the form. Each component (including the root form or data module) has an associated *IOTAComponent* interface. A wizard can examine or change most of the component's properties. If you need complete control over the component, you can cast the *IOTAComponent* interface to *INTAComponent*. The native component interface enables your wizard to access the *TComponent* pointer directly. This is important if you need to read or modify a class-type property, such as *TFont*, which is possible only through NTA-style interfaces.

Creating Forms and Projects

Delphi comes with a number of form and project wizards already installed, and you can write your own. The Object Repository lets you create static templates that can be used in a project, but a wizard offers much more power because it is dynamic. The wizard can prompt the user and create different kinds of files depending on the user's responses.

A form or project wizard typically creates one or more new files. Instead of real files, however, it is best to create unnamed, unsaved modules. When the user saves them, the IDE prompts the user for a file name. A wizard uses a creator object to create such modules.

A creator class implements a creator interface, which inherits from *IOTACreator*. The wizard passes a creator object to the module service's *CreateModule* method, and the IDE calls back to the creator object for the parameters it needs to create the module.

For example, a form wizard that creates a new form typically implements *GetExisting* to return *false* and *GetUnnamed* to return *true*. This creates a module that has no name (so the user must pick a name before the file can be saved) and is not backed by an existing file (so the user must save the file even if the user does not make any changes). Other methods of the creator tell the IDE what kind of file is being created (e.g., project, unit, or form), provide the contents of the file, or return the form name, ancestor name, and other important information. Additional callbacks let a wizard add modules to a newly created project, or add components to a newly created form.

To create a new file, which is often required in a form or project wizard, you usually need to provide the contents of the new file. To do so, write a new class that implements the *IOTAFile* interface. If your wizard can make do with the default file contents, you can return **nil** from any function that returns *IOTAFile*.

For example, suppose your organization has a standard comment block that must appear at the top of each source file. You could do this with a static template in the Object Repository, but the comment block would need to be

updated manually to reflect the author and creation date. Instead, you can use a creator to dynamically fill in the comment block when the file is created.

The first step is to write a wizard that creates new units and forms. Most of the creator's functions return zero, empty strings, or other default values, which tells the Tools API to use its default behavior for creating a new unit or form. Override *GetCreatorType* to inform the Tools API what kind of module to create: a unit or a form. To create a unit, return sUnit. To create a form, return sForm. To simplify the code, use a single class that takes the creator type as an argument to the constructor. Save the creator type in a data member, so that *GetCreatorType* can return its value. Implement *NewImplSource* and *NewIntfSource* to return the desired file contents.

```
[Delphi]
TCreator = class(TInterfacedObject, IOTAModuleCreator)
public
constructor Create(const CreatorType: string);
  { IOTAModuleCreator }
  function GetAncestorName: string;
  function GetImplFileName: string;
  function GetIntfFileName: string;
  function GetFormName: string;
  function GetMainForm: Boolean;
  function GetShowForm: Boolean;
  function GetShowSource: Boolean;
  function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
  function NewImplSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
  function NewIntfSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
  procedure FormCreated(const FormEditor: IOTAFormEditor);
  { IOTACreator }
  function GetCreatorType: string;
  function GetExisting: Boolean;
  function GetFileSystem: string;
  function GetOwner: IOTAModule;
  function GetUnnamed: Boolean;
private
  FCreatorType: string;
end;
```

```
[C++]
class PACKAGE Creator : public IOTAModuleCreator {
public:
__fastcall Creator(const AnsiString creator_type)
: ref_count(0), creator_type(creator_type) {}
virtual __fastcall ~Creator();
// IOTAModuleCreator
virtual AnsiString __fastcall GetAncestorName();
virtual AnsiString __fastcall GetImplFileName();
virtual AnsiString __fastcall GetIntfFileName();
virtual AnsiString __fastcall GetFormName();
virtual bool __fastcall GetMainForm();
virtual bool __fastcall GetShowForm();
virtual bool __fastcall GetShowSource();
virtual _di_IOTAFile __fastcall NewFormFile(
const AnsiString FormIdent, const AnsiString AncestorIdent);
virtual _di_IOTAFile __fastcall NewImplSource(
const AnsiString ModuleIdent, const AnsiString FormIdent,
const AnsiString AncestorIdent);
virtual _di_IOTAFile __fastcall NewIntfSource(
const AnsiString ModuleIdent, const AnsiString FormIdent,
const AnsiString AncestorIdent);
virtual void __fastcall FormCreated(
```

```
const _di_IOTAFormEditor FormEditor);
// IOTACreator
virtual AnsiString __fastcall GetCreatorType();
 virtual bool __fastcall GetExisting();
virtual AnsiString __fastcall GetFileSystem();
virtual _di_IOTAModule __fastcall GetOwner();
virtual bool __fastcall GetUnnamed();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
virtual ULONG __stdcall Release();
private:
long ref_count;
const AnsiString creator_type;
};
```

Most of the members of *T*Creator return zero, **nil**, or empty strings. The boolean methods return *true*, except *GetExisting*, which returns *false*. The most interesting method is *GetOwner*, which returns a pointer to the current project module, or **nil** if there is no project. There is no simple way to discover the current project or the current project group. Instead, *GetOwner* must iterate over all open modules. If a project group is found, it must be the only project group open, so *GetOwner* returns its current project. Otherwise, the function returns the first project module it finds, or **nil** if no projects are open.

```
[Delphi]
function TCreator.GetOwner: IOTAModule;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Project: IOTAProject;
  Group: IOTAProjectGroup;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Result := nil;
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProject, Project) then
    begin
      { Remember the first project module}
      if Result = nil then
        Result := Project;
    end
    else if Supports(Module, IOTAProjectGroup, Group) then
    begin
      { Found the project group, so return its active project}
      Result := Group.ActiveProject;
      Exit;
    end;
  end;
end;
```

```
[C++]
_di_IOTAModule __fastcall Creator::GetOwner()
{
// Return the current project.
_di_IOTAProject result = 0;
```

```
  _di_IOTAModuleServices svc = interface_cast<IOTAModuleServices>(BorlandIDEServices);
  for (int i = 0; i < svc->ModuleCount; ++i)
  begin
    _di_IOTAModule module = svc->Modules[i];
_di_IOTAProject project;
_di_IOTAProjectGroup group;
if (module->Supports(project)) {
// Remember the first project module.
if (result == 0)
result = project;
} else if (module->Supports(group)) {
// Found the project group, so return its active project.
result = group->ActiveProject;
break;
}
}
return result;
}
```

The creator returns **nil** from *NewFormSource*, to generate a default form file. The interesting methods are *NewImplSource* and *NewIntfSource*, which create an *IOTAFile* instance that returns the file contents.

The *TFile* class implements the *IOTAFile* interface. It returns -1 as the file age (which means the file does not exist), and returns the file contents as a string. To keep the *TFile* class simple, the creator generates the string, and the *TFile* class simply passes it on.

```
[Delphi]
TFile = class(TInterfacedObject, IOTAFile)
public
constructor Create(const Source: string);
  function GetSource: string;
  function GetAge: TDateTime;
private
  FSource: string;
end;
constructor TFile.Create(const Source: string);
begin
  FSource := Source;
end;
  function TFile.GetSource: string;
begin
  Result := FSource;
end;
  function TFile.GetAge: TDateTime;
begin
  Result := TDateTime(-1);
end;
```

```
[C++]
class File : public IOTAFile {
public:
__fastcall File(const AnsiString source);
virtual __fastcall ~File();
AnsiString __fastcall GetSource();
System::TDateTime __fastcall GetAge();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
```

```
virtual ULONG __stdcall Release();
private:
long ref_count;
AnsiString source;
};
__fastcall File::File(const AnsiString source)
: ref_count(0), source(source)
{}
AnsiString __fastcall File::GetSource()
{
return source;
}
System::TDateTime __fastcall File::GetAge()
{
return -1;
}
```

You can store the text for the file contents in a resource to make it easier to modify, but for the sake of simplicity, this example hardcodes the source code in the wizard. The example below generates the source code, assuming there is a form. You can easily add the simpler case of a plain unit. Test *FormIdent*, and if it is empty, create a plain unit; otherwise create a form unit. The basic skeleton for the code is the same as the IDE's default (with the addition of the comments at the top, of course), but you can modify it any way you desire.

```
[Delphi]
function TCreator.NewImplSource(
                    const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
var
  FormSource: string;
begin
  FormSource :=
  '{ ------------------------------------------------------------------- ' +   #13#10 +
  '%s - description'+   #13#10 +
  'Copyright ©  %y Your company, inc.'+   #13#10 +
  'Created on %d'+   #13#10 +
  'By %u'+   #13#10 +
  ' ------------------------------------------------------------------- }' +   #13#10 +
#13#10;
  return TFile.Create(Format(FormSource, ModuleIdent, FormIdent,
AncestorIdent));
}
```

```
[C++]
_di_IOTAFile __fastcall Creator::NewImplSource(
const AnsiString ModuleIdent,
const AnsiString FormIdent,
const AnsiString AncestorIdent)
{
const AnsiString form_source =
"/*-------------------------------------------------------------------\n"
" %m - description\n"
" Copyright \xa9  %y Your company, inc.\n"
" Created on %d\n"
" By %u\n"
"  -------------------------------------------------------------------*/\n"
"\n"
"#include <vcl.h>\n"
"#pragma hdrstop\n"
"\n"
"#include \"%m.h\"\n"
```

```
"//---------------------------------------------------------------------------\n"
"#pragma package(smart_init)\n"
"#pragma resource \"*.dfm\"\n"
"T%f *%f;\n"
"//---------------------------------------------------------------------------\n"
"__fastcall T%m::T%m(TComponent* Owner)\n"
"        : T%a(Owner)\n"
"{\n"
"}\n"
"//---------------------------------------------------------------------------\n";
  return new File(expand(form_source, ModuleIdent, FormIdent,
AncestorIdent));
}
```

The final step is to create two form wizards: one uses sUnit as the creator type, and the other uses sForm. As an added benefit for the user, you can use *INTAServices* to add a menu item to the **File** ▸ **New** menu to invoke each wizard. The menu item's *OnClick* event handler can call the wizard's *Execute* function.

Some wizards need to enable or disable the menu items, depending on what else is happening in the IDE. For example, a wizard that checks a project into a source code control system should disable its **Check In** menu item if no files are open in the IDE. You can add this capability to your wizard by using notifiers.

Notifying a Wizard of IDE Events

An important aspect of writing a well-behaved wizard is to have the wizard respond to IDE events. In particular, any wizard that keeps track of module interfaces must know when the user closes the module, so the wizard can release the interface. To do this, the wizard needs a notifier, which means you must write a notifier class.

All notifier classes implement one or more notifier interfaces. The notifier interfaces define callback methods; the wizard registers a notifier object with the Tools API, and the IDE calls back to the notifier when something important happens.

Every notifier interface inherits from *IOTANotifier*, although not all of its methods are used for a particular notifier. The following table lists all the notifier interfaces, and gives a brief description of each one.

Notifier interfaces

Interface	Description
IOTANotifier	Abstract base class for all notifiers
IOTABreakpointNotifier	Triggering or changing a breakpoint in the debugger
IOTADebuggerNotifier	Running a program in the debugger, or adding or deleting breakpoints
IOTAEditLineNotifier	Tracking movements of lines in the source editor
IOTAEditorNotifier	Modifying or saving a source file, or switching files in the editor
IOTAFormNotifier	Saving a form, or modifying the form or any components on the form (or data module)
IOTAIDENotifier	Loading projects, installing packages, and other global IDE events
IOTAMessageNotifier	Adding or removing tabs (message groups) in the message view
IOTAModuleNotifier	Changing, saving, or renaming a module
IOTAProcessModNotifier	Loading a process module in the debugger
IOTAProcessNotifier	Creating or destroying threads and processes in the debugger
IOTAThreadNotifier	Changing a thread's state in the debugger
IOTAToolsFilterNotifier	Invoking a tools filter

To see how to use notifiers, consider the example in Creating forms and projects. Using module creators, the example creates a wizard that adds a comment to each source file. The comment includes the unit's initial name, but the user almost always saves the file under a different name. In that case, it would be a courtesy to the user if the wizard updated the comment to match the file's true name.

To do this, you need a module notifier. The wizard saves the module interface that *CreateModule* returns, and uses it to register a module notifier. The module notifier receives notification when the user modifies the file or saves the file, but these events are not important for this wizard, so the *AfterSave* and related functions all have empty bodies. The important function is *ModuleRenamed*, which the IDE calls when the user saves the file under a new name. The declaration for the module notifier class is shown below:

```
[Delphi]
TModuleIdentifier = class(TNotifierObject, IOTAModuleNotifier)
public
constructor Create(const Module: IOTAModule);
  destructor Destroy; override;
  function CheckOverwrite: Boolean;
  procedure ModuleRenamed(const NewName: string);
  procedure Destroyed;
private
  FModule: IOTAModule;
  FName: string;
  FIndex: Integer;
end;
```

```
[C++]
class ModuleNotifier : public NotifierObject, public IOTAModuleNotifier
{
typedef NotifierObject inherited;
public:
__fastcall ModuleNotifier(const _di_IOTAModule module);
__fastcall ~ModuleNotifier();
// IOTAModuleNotifier
virtual bool __fastcall CheckOverwrite();
virtual void __fastcall ModuleRenamed(const AnsiString NewName);
// IOTANotifier
void __fastcall AfterSave();
void __fastcall BeforeSave();
void __fastcall Destroyed();
void __fastcall Modified();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
virtual ULONG __stdcall Release();
private:
_di_IOTAModule module;
AnsiString name;        // Remember the module's old name.
int index;              // Notifier index.
};
```

One way to write a notifier is to have it register itself automatically in its constructor. The destructor unregisters the notifier. In the case of a module notifier, the IDE calls the *Destroyed* method when the user closes the file. In that case, the notifier must unregister itself and release its reference to the module interface. The IDE releases its reference to the notifier, which reduces its reference count to zero and frees the object. Therefore, you need to write the destructor defensively: the notifier might already be unregistered.

```
[Delphi]
constructor TModuleNotifier.Create( const Module: IOTAModule);
begin
  FIndex := -1;
  FModule := Module;
  { Register this notifier. }
  FIndex := Module.AddNotifier(self);
  { Remember the module's old name. }
  FName := ChangeFileExt(ExtractFileName(Module.FileName), '');
end;
destructor TModuleNotifier.Destroy;
begin
  { Unregister the notifier if that hasn't happened already. }
  if Findex >= 0 then
    FModule.RemoveNotifier(FIndex);
end;
procedure TModuleNotifier.Destroyed;
begin
   { The module interface is being destroyed, so clean up the notifier. }
  if Findex >= 0 then
  begin
    { Unregister the notifier. }
    FModule.RemoveNotifier(FIndex);
    FIndex := -1;
  end;
  FModule := nil;
end;
```

```
[C++]
__fastcall ModuleNotifier::ModuleNotifier(const _di_IOTAModule module)
: index(-1), module(module)
{
// Register this notifier.
index = module->AddNotifier(this);
// Remember the module's old name.
name = ChangeFileExt(ExtractFileName(module->FileName), "");
}
__fastcall ModuleNotifier::~ModuleNotifier()
{
// Unregister the notifier if that hasn't happened already.
if (index >= 0)
module->RemoveNotifier(index);
}
void __fastcall ModuleNotifier::Destroyed()
{
// The module interface is being destroyed, so clean up the notifier.
if (index >= 0)
{
// Unregister the notifier.
module->RemoveNotifier(index);
index = -1;
}
module = 0;
}
```

The IDE calls back to the notifier's *ModuleRenamed* function when the user renames the file. The function takes the new name as a parameter, which the wizard uses to update the comment in the file. To edit the source buffer, the wizard uses an edit position interface. The wizard finds the right position, double checks that it found the right text, and replaces that text with the new name.

```
[Delphi]
procedure TModuleNotifier.ModuleRenamed(const NewName: string);
var
  ModuleName: string;
  I: Integer;
  Editor: IOTAEditor;
  Buffer: IOTAEditBuffer;
  Pos: IOTAEditPosition;
  Check: string;
begin
  { Get the module name from the new file name. }
  ModuleName := ChangeFileExt(ExtractFileName(NewName), '');
for I := 0 to FModule.GetModuleFileCount - 1 do
  begin
   { Update every source editor buffer. }
    Editor := FModule.GetModuleFileEditor(I);
    if Supports(Editor, IOTAEditBuffer, Buffer) then
    begin
      Pos := Buffer.GetEditPosition;
{ The module name is on line 2 of the comment.
        Skip leading white space and copy the old module name,
        to double check we have the right spot. }
      Pos.Move(2, 1);
      Pos.MoveCursor(mmSkipWhite or mmSkipRight);
Check := Pos.RipText('', rfIncludeNumericChars or rfIncludeAlphaChars);
if Check = FName then
      begin
        Pos.Delete(Length(Check));    // Delete the old name.
        Pos.InsertText(ModuleName);   // Insert the new name.
        FName := ModuleName;          // Remember the new name.
      end;
    end;
  end;
end;
```

```
[C++]
void __fastcall ModuleNotifier::ModuleRenamed(const AnsiString NewName)
{
// Get the module name from the new file name.
AnsiString ModuleName = ChangeFileExt(ExtractFileName(NewName), "");
for (int i = 0; i < module->GetModuleFileCount(); ++i)
{
// Update every source editor buffer.
_di_IOTAEditor editor = module->GetModuleFileEditor(i);
_di_IOTAEditBuffer buffer;
if (editor->Supports(buffer))
{
_di_IOTAEditPosition pos = buffer->GetEditPosition();
// The module name is on line 2 of the comment.
// Skip leading white space and copy the old module name,
// to double check we have the right spot.
pos->Move(2, 1);
pos->MoveCursor(mmSkipWhite | mmSkipRight);
AnsiString check = pos->RipText("", rfIncludeNumericChars | rfIncludeAlphaChars);
if (check == name)
{
pos->Delete(check.Length());    // Delete the old name.
pos->InsertText(ModuleName);    // Insert the new name.
name = ModuleName;              // Remember the new name.
```

```
}
}
}
}
```

What if the user inserts additional comments above the module name? In that case, you need to use an edit line notifier to keep track of the line number where the module name sits. To do this, use the *IOTAEditLineNotifier* and *IOTAEditLineTracker* interfaces.

You need to be cautious when writing notifiers. You must make sure that no notifier outlives its wizard. For example, if the user were to use the wizard to create a new unit, then unload the wizard, there would still be a notifier attached to the unit. The results would be unpredictable, but most likely, the IDE would crash. Thus, the wizard needs to keep track of all of its notifiers, and must unregister every notifier before the wizard is destroyed. On the other hand, if the user closes the file first, the module notifier receives a *Destroyed* notification, which means the notifier must unregister itself and release all references to the module. The notifier must remove itself from the wizard's master notifier list, too.

Below is the final version of the wizard's *Execute* function. It creates the new module, uses the module interface and creates a module notifier, then saves the module notifier in an interface list (*TInterfaceList*).

```
[Delphi]
procedure DocWizard.Execute;
var
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Notifier: IOTAModuleNotifier;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Module := Svc.CreateModule(TCreator.Create(creator_type));
  Notifier := TModuleNotifier.Create(Module);
list.Add(Notifier);
end
```

```
[C++]
void __fastcall DocWizard::Execute()
{
_di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
_di_IOTAModule module = svc->CreateModule(new Creator(creator_type));
_di_IOTAModuleNotifier notifier = new ModuleNotifier(module);
list->Add(notifier);
}
```

The wizard's destructor iterates over the interface list and unregisters every notifier in the list. Simply letting the interface list release the interfaces it holds is not sufficient because the IDE also holds the same interfaces. You must tell the IDE to release the notifier interfaces in order to free the notifier objects. In this case, the destructor tricks the notifiers into thinking their modules have been destroyed. In a more complicated situation, you might find it best to write a separate Unregister function for the notifier class.

```
[Delphi]
destructor DocWizard.Destroy; override;
var
  Notifier: IOTAModuleNotifier;
  I: Integer;
begin
  { Unregister all the notifiers in the list. }
for I := list.Count - 1 downto 0 do
```

```
  begin
    Supports(list.Items[I], IOTANotifier, Notifier);
    { Pretend the associated object has been destroyed.
      That convinces the notifier to clean itself up. }
    Notifier.Destroyed;
    list.Delete(I);
  end;
  list.Free;
  FItem.Free;
end;
```

```
[C++]
__fastcall DocWizard::~DocWizard()
{
// Unregister all the notifiers in the list.
for (int i = list->Count; --i >= 0; )
{
_di_IOTANotifier notifier;
    list->Items[i]->Supports(notifier);
// Pretend the associated object has been destroyed.
// That convinces the notifier to clean itself up.
notifier->Destroyed();
list->Delete(i);
}
delete list;
delete item;
}
```

The rest of the wizard manages the mundane details of registering the wizard, installing menu items, and the like.

Index

.NET data types
  BDP.NET, 885
ADO.NET
  ASP.NET, 812
  Adapter Preview Editor, 1065 1065
  architecture, 877
  Command Text Editor, 1059 1059
  CommandText Editor, 1063 1063
  database applications, 1169 1169
  namespace, 879
  Windows Forms, 850
ADO.NET application, 955
  building, 1017 1017 1017
ASP.NET
  architecture, 811
  DB Web Controls, 949 949 949
ASP.NET application, 953
ASP.NET errors
  HTTP messages, 971 971
ASP.NET lifecycle
  ASP.NET processing, 827
Assembly Metadata Explorer, 177
audit, 95
BDP.NET
  data providers, 882
BDP.NET components
  BDP.NET, 882
bitmap images, 1191
bookmarks, using, 349
browsing a database, 1009
build configurations
  inherit checkbox, 1263
  default check box, 1263
  overriding inherited options, 1263
building
  VCL Forms hello world, 1167 1167
  VCL forms applications with XML components, 1171
    1171
  VCL Forms menus, 1193 1193
Caliber
  Requirements, 49
callback functions
  functions, 923
cascading deletes
  database, 823
cascading updates
  database, 824
ClearSessionChanges method, 819
code
  templates, 333 357 333 357
Code Editor
  customizing, 335 335
code folding, 353
CodeGuard
  using, 1283 1283
code insight, 355
Code Template, 91
COM interfaces
  interfaces, 925
COM Interop
  Terminology, 913
  Interop assemblies in the IDE, 916
  SDK Tools, 915
comment blocks, 40
component designers
  relationship, 889
  Command Text Editor, 890
  configure data adapter, 891
  Connection Editor, 890
  Dataset, 891
  Stored Procedure Dialog, 890
components
  data-aware, 815
  connection, 816
  importing, 857
  Windows Forms, 849
component templates, 165
constructors
  destructors, 868
data
  remoting, 1013 1013
  migration, 1041 1041
data-aware controls
  controls, 1773
database
  connections, 973 973 973
data binding
  DB Web Control binding, 827
data explorer
  definition, 891
Data Explorer
  executing SQL, 1037 1037
  modifying connections, 1043 1043
data providers
  architecture, 881
DataSet
  table mappings, 1035 1035
data types
  BDP.NET, 883
DataView limitations
  inserting records, 819

DataViews
runtime properties, 819
DB2
BDP.NET, 885
DB Web Controls
architecture, 815
library, 995 995 995
namespace, 816
DB Web interfaces, 829
debugging
adding a watch, 287 287
attaching to a process, 289 289
breakpoints, 321 321
inspecting data elements, 301 301
modifying expressions, 305 305
preparation, 293 307 293 307
remote applications, 291 291
Web Application Debugger, 1228
declare field, 115
declare variable
initial type, 115
declare variable and field samples, 116
declare variable rules, 115
Delphi for .NET
Web Forms, 811
Windows Forms, 849
deploying
BDP.NET applications, 909
ASP.NET Deployment Manager, 991 991
BDE applications for .NET, 910
dbExpress applications for .NET, 909
dbGo applications for .NET, 910
deployment
DB Web Controls, 816
Design Pattern, 91
displaying bitmap images, 1181
doc comment, 85
documentation generation, 97
drawing
polygons, 1183 1183 1405 1405
rectangles and ellipses, 1185 1185 1409 1409
round rectangles, 1407 1407
straight lines, 1187 1187 1411 1411
dynamic properties, 193
ECO
ECO terminology, 735
Behavioral Modeling with state machines, 741
Building Applications with the ECO Framework, 1103 1103
Chained evaluation, 749
Converting to ECO 3, 1113 1113
Creating an ECO space subclass, 1115 1115
Deployment, 1133 1133
Designtime functionality, 736
EcoSpaceProvider, 787
Interaction with Model View, 740
Pooling, 788
Predefined OCL Types, 777
Projects and wizards, 739
Reevaluate and Resubscribe, 755
Regenerating source code, 1145 1145
Regions of a state, 767
Rooted handles, 751
Root handles, 750
Runtime functionality, 737
Shared Persistence Mappers, 789
State machine basics, 763
Structural modeling with class diagram, 740
Subclassing SubscriberAdapterBase, 1141 1141
SubscriberAdapterBase abstract class, 758
Subscription Mechanism, 755
Subscriptions and derived attributes, 757
Synchronizing ECO Spaces, 789
Syntax of OCL Expressions, 778
ECO class diagram
Derived association end, 1073 1073
Association class, 1121 1121
Derived attribute, 1075 1075
Deriving attributes in source code, 1135 1135
ECO handles
Columns and nestings, 1097 1097
Configuring OclVariables, 1107 1107
Event derived columns, 1131 1131
ECO multi-client
Creating an ECO ASP.NET application, 1123 1123
ECO Object Constraint Language
OCL expression editor, 1153 1153
ECO packages
Adding a reference to an ECO package DLL, 1083 1083
Adding a UML package to a project, 1091 1091
ECO package in a DLL, 1127 1127
ECO space designer
Adding a PersistenceMapperClient, 1079 1079
Adding a Connection Handle, 1095 1095
Adding a PersistenceMapperSharer, 1081 1081
Configuring the Persistence Method, 1111 1111
Creating an Empty Interbase Database, 1129 1129
Mulitple data sources, 1105 1105
Object-relational mapping files, 1149 1149
Persistence mapper provider, 1119 1119
PersistenceMapperProvider Designer, 1155 1155
Reverse engineering an existing database, 1139 1139
Selecting ECO UML Packages, 1147 1147
Using the ECO space designer, 1151 1151

ECO state machines
Guard expressions, 1077 1077
Adding a Region to a State, 1085 1085
Adding a Trigger Method, 1087 1087
Adding States and Substates, 1101 1101
Creating a state machine, 1125 1125
Effects, 1093 1093
Entry and exit actions, 1099 1099
ECO Winforms
Adding an ECO enabled Windows Form, 1089 1089
ECO winforms
Creating a new ECO Windows Forms application, 1117 1117
editing code
class completion, 351 351
Evaluate/Modify
debugging, 145
expandable tooltips, 327
find references sample, 119
getcurdir, 1276
graphics
VCL Forms applications, 1373 1373
hello world
VCL Forms, 1341 1379 1427 1341 1379 1427
help
Delphi for .NET, 45
.NET Framework, 46
borland site, 46
quick start guide, 46
typographic, 46
History Manager, 361
HTML tag editor
editing HTML tags, 997 997
IDBWebColumnLink
IDBWebDataLink, 821
IDBWebLookupColumnLink, 821
IDBWebDataLink, 821
ide
welcome page, 27
Code Editor, 32
design surface, 28
forms, 27
MS Active Accessibility, 27
Object Inspector, 29
object repository, 30
Project Manager, 30
tool palette, 29
IIS
troubleshooting, 987 987
InterBase
BDP.NET, 886
components, 903
InterBase components
getting started, 1241
Interfaces
BDP.NET, 883
internal errors, 319
Janeva
installing to Tools menu, 1161 1161
logical data types
BDP.NET, 885
macro
recording, 341 341
master-detail
DataViews, 819
MDA, 85
menus
VCL Forms, 1383 1387 1403 1431 1383 1387 1403 1431
metadata
DataSet mappings, 967 967 967
method rename, 109
metric, 95
modal
VCL Forms, 1413 1413
modeless
VCL Forms, 1415 1415
modifying DB Web controls
extending controls, 829
MS SQL
BDP.NET, 886
multithreaded applications
simultaneous thread access, 1337 1337
cleaning up threads, 1439 1439
exception handling, 1417 1417
initializing threads, 1419 1419
main thread, 1433 1433
thread object, 1391 1391
waiting for threads, 1435 1435
writing the thread function, 1441 1441
name mapping, 87
New Language Features, 859
object references
passing references, 924
OCL, 89
constraint, 89
supported diagram types, 89
Oracle
BDP.NET, 887
override Render
Render, 828
packages

units, 273 273 273
parameters
database, 1051 1051 1051
Pascal
language changes, 859
Pattern Organizer, 91
Pattern Registry, 91
Patterns as First Class Citizens, 92
adding participants, 615 615
placing bitmap images, 1425
pointer types
porting, 861
porting
VCL.NET porting, 857
projects
types of, 33
additional projects, 35
radio buttons
data-aware, 1778
Rave Reports
creating new reports, 937
creating reports, 1279
Tools, 1333 1333
refactoring
preview, 279 279 279 279
procedures, 313 313 313
references
projects, 157 157
rename
renaming, 1429 1429
rename symbol
refactoring, 281 281 281 281
resolving
multiple tables, 1011 1011
resources
migrating, 872
sample
ASP.NET hello world, 951 951
sets, 96
socket components
Windows socket objects, 2181
source control, 53
StarTeam
embedded client, 53
active process item, 389 389
added features, 54
adding files, 377 377
advanced features, 54
checking in files, 379 379
checking out files, 381 381
comparing file revisions, 385 385
configuring, 387 387
finding files, 391 391
locking and unlocking files, 395 395
merging files, 397 397
migrating from SCC, 399 399
placing projects, 401 401
pulling projects from, 403 403
removing files, 405 405
reverting files, 407 407
updating projects, 383 409 383 409
version control support, 54
StarTeam Client
launching, 393 393
string
char, 865
strings
sort list, 1335 1335
adding and sorting, 1375 1389 1395 1421 1375 1389 1395 1421
structures
pointers, 921
stub implementation, 92
Sybase
BDP.NET, 887
Sync Edit, 359
table mappings
errors, 1039 1039
to-do lists
overview, 40
planning, 207 207
toolbars
customizing, 173 173
Tool Palette
components, 155 155
transform to source code, 87
translation tools
adding languages to a project, 365 365
editing with Translation Manager, 367 367
External Translation Manager, 375 375
setting the active language, 369 369
setting up the External Translation Manager, 371 371
typecasts
crackers, 866
UML, 71
UML In Color, 71
units
linking, 277 277 277
unit tests, 727
unmanaged code, 35
unmanaged functions
Win32 API, 919

update objects
executing statements, 1956
user interfaces
ADO.NET, 878
variants
TVarRec, 871
VCL
Architecture, 1249
VCL.NET
architecture, 855
namespace, 857
porting, 872
VCL.NET components
VCL.NET, 856
VCL applications
forms, 1163 1163
dbExpress, 1351 1351
dbExpress database applications, 1165 1165
graphics, 1175 1175
VCL Forms, 1345 1345
VCL forms
menus, 1179 1179
VCL Forms
multithreaded applications, 1339 1339
ActiveX Active Forms, 1385 1385
ADO.NET database applications, 1343 1343
decision support, 1347 1347
MDI applications, 1353 1355 1353 1355
SDI applications, 1359 1359
XML components, 1367 1367
VCL versus VCL.NET, 1249
virtual directory, creating in IDE, 963
Web Application Support
Win32, 1227
WebBroker, 1227
web browser
VCL Forms, 1361 1361
Web modules
data modules, 2097
web service clients
porting, 860
Web Services
ASP.NET, 812
client, 1195 1195
web services
ASP.NET, 812
architecture, 839
client support, 845
discovery, 844
files, 841
namespaces, 846
porting Win32 to .NET, 979 979 979
prerequisites, 839
protocol, 843
scenarios, 840
server support, 846
service description, 844
service transport, 844
web references, 943 943 943
xml messaging, 844
WebSnap, 1227
hello world, 1443 1443
Win32 API
unmanaged code, 868
Windows Forms
architecture, 849
hello world, 1363 1363
menus, 1215 1215
namespace, 850
Windows Forms application, 1207
building, 1365 1365
Windows Forms hello world
building, 1213 1213
XML and authentication
XML caching, 825
XML files
XML advantages, 825
DBWeb XML files, 965 965 965