



Developer's Guide



Borland®
Delphi™ 5
for Windows 98, Windows 95, & Windows NT

Inprise Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Refer to the file DEPLOY.TXT located in the root directory of your Delphi 5 product for a complete list of files that you can distribute in accordance with the Delphi 5 License Statement and Limited Warranty.

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 1999 Inprise Corporation. All rights reserved. All Inprise and Borland brand and product names are trademarks or registered trademarks of Inprise Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

HDE1350WW21001 3E2R799

9900010203-9 8 7 6 5 4 3 2 1

PDF

Contents

Chapter 1

Introduction 1-1

What's in this manual?	1-1
Manual conventions	1-2
Developer support services.	1-3
Ordering printed documentation	1-3

Part I

Programming with Delphi

Chapter 2

Using Object Pascal with the VCL 2-1

Object Pascal and the VCL	2-1
Using the object model	2-2
What is an object?	2-2
Examining a Delphi object	2-2
Inheriting data and code from an object	2-5
Objects, components, and controls	2-6
Scope and qualifiers	2-6
Private, protected, public, and published declarations	2-7
Using object variables	2-7
Creating, instantiating, and destroying objects	2-8
Components and ownership	2-9
Using components	2-10
C++Builder's standard components	2-10
Properties common to visual components	2-11
Text controls	2-12
Specialized input controls	2-13
Buttons and similar controls	2-15
Handling lists	2-17
Grouping components	2-18
Visual feedback	2-20
Grids	2-21
Graphic display	2-22
Windows common dialog boxes	2-23
Setting component properties.	2-23
Using the Object Inspector.	2-23
Setting properties at runtime	2-24
Calling methods	2-24
Working with events and event handlers	2-24
Generating a new event handler	2-24
Generating a handler for a component's default event.	2-25

Locating event handlers.	2-25
Associating an event with an existing event handler.	2-25
Associating menu events with event handlers.	2-26
Deleting event handlers.	2-27
Using helper objects	2-27
Working with lists.	2-27
Working with string lists	2-28
Loading and saving string lists.	2-28
Creating a new string list	2-28
Manipulating strings in a list	2-30
Associating objects with a string list.	2-32
The Windows registry and INI files	2-32
Using streams	2-33
Using data modules and remote data modules	2-33
Creating and editing data modules	2-33
Creating business rules in a data module	2-34
Accessing a data module from a form	2-34
Adding a remote data module to an application server project.	2-34
Using the Object Repository	2-35
Sharing items within a project	2-35
Adding items to the Object Repository	2-35
Sharing objects in a team environment	2-35
Using an Object Repository item in a project.	2-36
Copying an item	2-36
Inheriting an item	2-36
Using an item.	2-36
Using project templates.	2-36
Modifying shared items	2-37
Specifying a default project, new form, and main form	2-37
Adding custom components to the IDE	2-37

Chapter 3

Common programming tasks 3-1

Handling exceptions	3-1
Protecting blocks of code	3-1
Responding to exceptions.	3-2
Exceptions and the flow of control.	3-2
Nesting exception responses	3-3
Protecting resource allocations.	3-4

What kind of resources need protection?	3-4	Passing a local variable as a PChar	3-31
Creating a resource protection block	3-5	Compiler directives for strings.	3-31
Handling RTL exceptions	3-5	Strings and characters: related topics	3-32
What are the RTL exceptions?	3-6	Working with files	3-32
Creating an exception handler	3-7	Manipulating files.	3-33
Exception handling statements	3-7	Deleting a file.	3-33
Using the exception instance	3-8	Finding a file	3-33
Scope of exception handlers	3-8	Changing file attributes.	3-35
Providing default exception handlers	3-9	Renaming a file.	3-35
Handling classes of exceptions	3-9	File date-time routines	3-35
Reraising the exception	3-10	Copying a file	3-36
Handling component exceptions	3-11	File types with file I/O	3-36
Using TApplication.HandleException	3-11	Using file streams	3-37
Silent exceptions	3-12	Creating and opening files	3-37
Defining your own exceptions	3-12	Using the file handle	3-38
Declaring an exception object type	3-13	Reading and writing to files	3-38
Raising an exception	3-13	Reading and writing strings	3-39
Using interfaces	3-14	Seeking a file	3-39
Interfaces as a language feature.	3-14	File position and size	3-39
Sharing interfaces between classes	3-15	Copying.	3-40
Using interfaces with procedures	3-16	Defining new data types	3-40
Implementing IUnknown	3-16		
TInterfacedObject	3-17	Chapter 4	
Using the as operator	3-17	Building applications, components,	
Reusing code and delegation	3-18	and libraries	4-1
Using implements for delegation	3-18	Creating applications	4-1
Aggregation	3-19	Windows applications	4-1
Memory management of interface objects.	3-20	User interface models	4-2
Using reference counting	3-20	Setting IDE, project, and compilation options	4-2
Not using reference counting	3-21	Programming templates	4-2
Using interfaces in distributed applications	3-22	Console applications	4-3
Working with strings	3-23	Service applications	4-3
Character types	3-23	Service threads	4-5
String types	3-23	Service name properties.	4-7
Short strings	3-24	Debugging services	4-8
Long strings	3-24	Creating packages and DLLs	4-8
WideString.	3-25	When to use packages and DLLs	4-8
PChar types	3-25	Writing database applications.	4-9
OpenString	3-25	Building distributed applications	4-9
Runtime library string handling routines	3-26	Distributing applications using TCP/IP	4-10
Wide character routines	3-26	Using sockets in applications.	4-10
Commonly used long string routines.	3-26	Creating Web server applications	4-10
Declaring and initializing strings.	3-28	Distributing applications using COM and DCOM	4-11
Mixing and converting string types	3-30	COM and DCOM	4-11
String to PChar conversions.	3-30	MTS	4-11
String dependencies	3-30	Distributing applications using CORBA	4-11
Returning a PChar local variable	3-30	Distributing database applications	4-12

Chapter 5

Developing the application user interface

5-1

- Understanding TApplication, TScreen, and TForm 5-1
 - Using the main form 5-1
 - Adding additional forms 5-2
 - Linking forms 5-2
 - Working at the application level 5-3
 - Handling the screen 5-3
 - Managing layout 5-3
- Working with messages 5-4
- More details on forms 5-5
 - Controlling when forms reside in memory 5-5
 - Displaying an auto-created form 5-5
 - Creating forms dynamically 5-5
 - Creating modeless forms such as windows 5-6
 - Using a local variable to create a form instance. 5-7
 - Passing additional arguments to forms 5-7
 - Retrieving data from forms 5-8
 - Retrieving data from modeless forms 5-8
 - Retrieving data from modal forms 5-9
- Reusing components and groups of components 5-11
- Creating and using component templates 5-12
- Working with frames 5-12
 - Creating frames. 5-13
 - Adding frames to the Component palette 5-13
 - Using and modifying frames 5-13
 - Sharing frames 5-14
- Creating and managing menus. 5-15
 - Opening the Menu Designer 5-15
 - Building menus. 5-17
 - Naming menus 5-17
 - Naming the menu items 5-17
 - Adding, inserting, and deleting menu items 5-18
 - Creating submenus 5-19
 - Viewing the menu 5-21
 - Editing menu items in the Object Inspector. 5-22
 - Using the Menu Designer context menu. 5-22
 - Commands on the context menu 5-23
 - Switching between menus at design time. 5-23

- Using menu templates 5-24
- Saving a menu as a template. 5-25
 - Naming conventions for template menu items and event handlers 5-26
- Manipulating menu items at runtime. 5-26
- Merging menus 5-26
 - Specifying the active menu: Menu property. 5-27
 - Determining the order of merged menu items: GroupIndex property. 5-27
 - Importing resource files 5-27
- Designing toolbars and cool bars 5-28
 - Adding a toolbar using a panel component 5-29
 - Adding a speed button to a panel 5-29
 - Assigning a speed button's glyph 5-29
 - Setting the initial condition of a speed button 5-30
 - Creating a group of speed buttons. 5-30
 - Allowing toggle buttons 5-30
 - Adding a toolbar using the toolbar component 5-31
 - Adding a tool button 5-31
 - Assigning images to tool buttons 5-31
 - Setting tool button appearance and initial conditions 5-32
 - Creating groups of tool buttons 5-32
 - Allowing toggled tool buttons 5-32
- Adding a cool bar component 5-33
 - Setting the appearance of the cool bar. 5-33
- Responding to clicks 5-34
 - Assigning a menu to a tool button. 5-34
- Adding hidden toolbars 5-34
- Hiding and showing toolbars 5-34
- Using action lists 5-35
- Action objects 5-35
- Using actions. 5-36
 - Centralizing code 5-37
 - Linking properties 5-37
 - Executing actions 5-37
 - Updating actions. 5-39
- Pre-defined action classes 5-39
 - Standard edit actions 5-39
 - Standard Window actions. 5-40
 - Standard Help actions. 5-40
 - DataSet actions 5-41
- Writing action components. 5-41
 - How actions find their targets 5-41
 - Registering actions. 5-43

Writing action list editors	5-43
Demo programs	5-43
Chapter 6	
Working with controls	6-1
Implementing drag-and-drop in controls	6-1
Starting a drag operation	6-1
Accepting dragged items	6-2
Dropping items	6-2
Ending a drag operation	6-3
Customizing drag and drop with a drag object	6-3
Changing the drag mouse pointer	6-4
Implementing drag-and-dock in controls	6-4
Making a windowed control a docking site	6-4
Making a control a dockable child	6-4
Controlling how child controls are docked	6-5
Controlling how child controls are undocked	6-6
Controlling how child controls respond to drag-and-dock operations	6-6
Working with text in controls	6-6
Setting text alignment	6-6
Adding scroll bars at runtime	6-7
Adding the Clipboard object	6-8
Selecting text	6-8
Selecting all text	6-8
Cutting, copying, and pasting text	6-9
Deleting selected text	6-9
Disabling menu items	6-9
Providing a pop-up menu	6-10
Handling the OnPopup event	6-11
Adding graphics to controls	6-11
Setting the owner-draw style	6-12
Adding graphical objects to a string list	6-12
Adding images to an application	6-12
Adding images to a string list	6-13
Drawing owner-drawn items	6-13
Sizing owner-draw items	6-13
Drawing each owner-draw item	6-14

Chapter 7	
Working with graphics and multimedia	7-1
Overview of graphics programming	7-1
Refreshing the screen	7-2
Types of graphic objects	7-2

Common properties and methods of Canvas	7-3
Using the properties of the Canvas object	7-5
Using pens	7-5
Using brushes	7-7
Reading and setting pixels	7-9
Using Canvas methods to draw graphic objects	7-9
Drawing lines and polylines	7-9
Drawing shapes	7-10
Handling multiple drawing objects in your application	7-11
Keeping track of which drawing tool to use	7-11
Changing the tool with speed buttons	7-12
Using drawing tools	7-13
Drawing on a graphic	7-16
Making scrollable graphics	7-16
Adding an image control	7-16
Loading and saving graphics files	7-18
Loading a picture from a file	7-18
Saving a picture to a file	7-19
Replacing the picture	7-19
Using the Clipboard with graphics	7-20
Copying graphics to the Clipboard	7-21
Cutting graphics to the Clipboard	7-21
Pasting graphics from the Clipboard	7-21
Rubber banding example	7-22
Responding to the mouse	7-22
Adding a field to a form object to track mouse actions	7-25
Refining line drawing	7-26
Working with multimedia	7-28
Adding silent video clips to an application	7-28
Example of adding silent video clips	7-29
Adding audio and/or video clips to an application	7-30
Example of adding audio and/or video clips	7-31

Chapter 8	
Writing multi-threaded applications	8-1
Defining thread objects	8-1
Initializing the thread	8-2
Assigning a default priority	8-2
Indicating when threads are freed	8-3

Writing the thread function	8-3
Using the main VCL thread	8-4
Using thread-local variables	8-5
Checking for termination by other threads	8-5
Writing clean-up code	8-6
Coordinating threads	8-6
Avoiding simultaneous access	8-6
Locking objects	8-6
Using critical sections	8-7
Using the multi-read exclusive-write synchronizer	8-7
Other techniques for sharing memory	8-8
Waiting for other threads	8-8
Waiting for a thread to finish executing	8-8
Waiting for a task to be completed	8-9
Executing thread objects	8-10
Overriding the default priority	8-10
Starting and stopping threads	8-10
Using threads in distributed applications	8-11
Using threads in message-based servers	8-11
Using threads with distributed objects	8-11
Writing applications (.EXEs)	8-11
Writing libraries	8-12
Debugging multi-threaded applications	8-13

Chapter 9

Working with packages and components

9-1

Why use packages?	9-2
Packages and standard DLLs	9-2
Runtime packages	9-2
Using packages in an application	9-3
Dynamically loading packages	9-3
Deciding which runtime packages to use	9-4
Custom packages	9-5
Design-time packages	9-5
Installing component packages	9-6
Creating and editing packages	9-7
Creating a package	9-7
Editing an existing package	9-8
Editing package source files manually	9-8
Understanding the structure of a package	9-9
Naming packages	9-9
The Requires clause	9-9
The Contains clause	9-10
Compiling packages	9-10
Package-specific compiler directives	9-11

Using the command-line compiler and linker	9-12
Package files created by a successful compilation	9-13
Deploying packages	9-13
Deploying applications that use packages	9-13
Distributing packages to other developers	9-13
Package collection files	9-13

Chapter 10

Creating international applications

10-1

Internationalization and localization	10-1
Internationalization	10-1
Localization	10-1
Internationalizing applications	10-2
Enabling application code	10-2
Character sets	10-2
OEM and ANSI character sets	10-2
Double byte character sets	10-2
Wide characters	10-3
Including bi-directional functionality in applications	10-4
BiDiMode property	10-5
Locale-specific features	10-7
Designing the user interface	10-8
Text	10-8
Graphic images	10-8
Formats and sort order	10-9
Keyboard mappings	10-9
Isolating resources	10-9
Creating resource DLLs	10-9
Using resource DLLs	10-11
Dynamic switching of resource DLLs	10-12
Localizing applications	10-12
Localizing resources	10-12

Chapter 11

Deploying applications

11-1

Deploying general applications	11-1
Using installation programs	11-2
Identifying application files	11-2
Application files, listed by file name extension	11-2
Package files	11-3
ActiveX controls	11-3
Helper applications	11-3
DLL locations	11-3
Deploying database applications	11-4

Providing the database engine	11-4
Borland Database Engine	11-4
Third-party database engines	11-5
SQL Links	11-5
Multi-tiered Distributed Application	
Services (MIDAS).	11-5
Deploying Web applications	11-7
Programming for varying host environments	11-7
Screen resolutions and color depths	11-7
Considerations when not dynamically resizing.	11-8
Considerations when dynamically resizing forms and controls	11-8
Accommodating varying color depths	11-9
Fonts	11-9
Windows versions	11-10
Software license requirements	11-10
DEPLOY.TXT	11-11
README.TXT	11-11
No-nonsense license agreement	11-11
Third-party product documentation	11-11

Part II

Developing database applications

Chapter 12

Designing database applications 12-1

Using databases	12-1
Types of databases	12-2
Local databases	12-2
Remote database servers	12-2
Database security	12-3
Transactions	12-3
Data Dictionary	12-4
Referential integrity, stored procedures, and triggers	12-5
Database architecture	12-6
Planning for scalability	12-6
Single-tiered database applications	12-8
Two-tiered database applications	12-8
Multi-tiered database applications	12-9
Designing the user interface	12-11
Displaying a single record	12-11
Displaying multiple records	12-12
Analyzing data	12-12
Selecting what data to show	12-13
Writing reports	12-15

Chapter 13

Building one- and two-tiered applications 13-1

BDE-based applications	13-2
BDE-based architecture	13-2
Understanding databases and datasets	13-3
Using sessions	13-4
Connecting to databases	13-5
Using transactions	13-5
Explicitly controlling transactions	13-6
Using a database component for transactions	13-6
Using the TransIsolation property	13-7
Using passthrough SQL	13-8
Using local transactions	13-9
Caching updates	13-9
Creating and restructuring database tables	13-10
ADO-based applications	13-10
ADO-based architecture	13-11
Understanding ADO databases and datasets	13-11
Connecting to ADO databases	13-12
Retrieving data	13-12
Creating and restructuring ADO database tables	13-13
Flat-file database applications	13-13
Creating the datasets	13-14
Creating a new dataset using persistent fields	13-14
Creating a dataset using field and index definitions	13-15
Creating a dataset based on an existing table	13-16
Loading and saving data	13-16
Using the briefcase model	13-17
Scaling up to a three-tiered application	13-18

Chapter 14

Creating multi-tiered applications 14-1

Advantages of the multi-tiered database model	14-2
Understanding MIDAS technology	14-2
Overview of a MIDAS-based multi-tiered application	14-3
The structure of the client application	14-4
The structure of the application server	14-4

Using MTS	14-5	Creating an Active Form for the client application	14-29
Pooling remote data modules	14-7	Building Web applications using InternetExpress	14-30
Using the IAppServer interface	14-7	Building an InternetExpress application	14-30
Choosing a connection protocol	14-8	Using the javascript libraries	14-32
Using DCOM connections	14-8	Granting permission to access and launch the application server	14-32
Using Socket connections	14-9	Using an XML broker	14-33
Using Web connections	14-9	Fetching XML data packets	14-33
Using OLEEnterprise	14-10	Applying updates from XML delta packets	14-34
Using CORBA connections	14-10	Creating Web pages with a MIDAS page producer	14-35
Building a multi-tiered application	14-10	Using the Web page editor	14-35
Creating the application server	14-11	Setting Web item properties	14-36
Setting up the remote data module	14-13	Customizing the MIDAS page producer template	14-37
Configuring TRemoteDataModule	14-13		
Configuring TMTSDDataModule	14-14		
Configuring TCorbaDataModule	14-15		
Creating a data provider for the application server	14-16		
Extending the application server's interface	14-16		
Adding callbacks to the application server's interface	14-17		
Extending the application server's interface when using MTS	14-17		
Creating the client application	14-18		
Connecting to the application server	14-19		
Specifying a connection using DCOM	14-19		
Specifying a connection using sockets	14-20		
Specifying a connection using HTTP	14-21		
Specifying a connection using OLEEnterprise	14-21		
Specifying a connection using CORBA	14-22		
Brokering connections	14-22		
Managing server connections	14-23		
Connecting to the server	14-23		
Dropping or changing a server connection	14-23		
Calling server interfaces	14-24		
Managing transactions in multi-tiered applications	14-25		
Supporting master/detail relationships	14-26		
Supporting state information in remote data modules	14-26		
Writing MIDAS Web applications	14-28		
Distributing a client application as an ActiveX control	14-29		
		Chapter 15	
		Using provider components	15-1
		Determining the source of data	15-1
		Choosing how to apply updates	15-2
		Controlling what information is included in data packets	15-2
		Specifying what fields appear in data packets	15-2
		Setting options that influence the data packets	15-3
		Adding custom information to data packets	15-4
		Responding to client data requests	15-5
		Responding to client update requests	15-5
		Editing delta packets before updating the database	15-6
		Influencing how updates are applied	15-7
		Screening individual updates	15-9
		Resolving update errors on the provider	15-9
		Applying updates to datasets that do not represent a single table	15-9
		Responding to client-generated events	15-10
		Handling server constraints	15-10
		Chapter 16	
		Managing database sessions	16-1
		Working with a session component	16-1
		Using the default session	16-2
		Creating additional sessions	16-3

Naming a session	16-4	Setting BDE alias parameters	17-5
Activating a session	16-4	Controlling server login	17-6
Customizing session start-up	16-5	Connecting to a database server	17-6
Specifying default database connection behavior	16-6	Special considerations when connecting to a remote server	17-7
Creating, opening, and closing database connections	16-6	Working with network protocols.	17-7
Closing a single database connection.	16-7	Using ODBC	17-8
Closing all database connections	16-7	Disconnecting from a database server	17-8
Dropping temporary database connections	16-8	Closing datasets without disconnecting from a server	17-8
Searching for a database connection	16-8	Iterating through a database component's datasets	17-8
Retrieving information about a session	16-8	Understanding database and session component interactions.	17-9
Working with BDE aliases.	16-9	Using database components in data modules	17-9
Specifying alias visibility.	16-10	Executing SQL statements from a TDatabase component.	17-9
Making session aliases visible to other sessions and applications	16-10	Executing SQL statements without result sets.	17-10
Determining known aliases, drivers, and parameters	16-10	Executing SQL statements with result sets	17-11
Creating, modifying, and deleting aliases	16-11	Executing parameterized SQL statements	17-12
Iterating through a session's database components	16-12		
Specifying Paradox directory locations	16-13		
Specifying the control file location	16-13		
Specifying a temporary files location.	16-13		
Working with password-protected Paradox and dBase tables	16-13		
Using the AddPassword method	16-14		
Using the RemovePassword and RemoveAllPasswords methods	16-14		
Using the GetPassword method and OnPassword event	16-15		
Managing multiple sessions	16-16		
Using a session component in data modules	16-17		
Chapter 17		Chapter 18	
Connecting to databases	17-1	Understanding datasets	18-1
Understanding persistent and temporary database components	17-1	What is TDataSet?.	18-2
Using temporary database components	17-2	Types of datasets	18-2
Creating database components at design time.	17-2	Opening and closing datasets	18-3
Creating database components at runtime	17-2	Determining and setting dataset states.	18-3
Controlling connections.	17-4	Inactivating a dataset	18-5
Associating a database component with a session	17-4	Browsing a dataset	18-6
Specifying a BDE alias	17-4	Enabling dataset editing	18-6
		Enabling insertion of new records.	18-7
		Enabling index-based searches and ranges on tables	18-8
		Calculating fields	18-8
		Filtering records	18-8
		Updating records	18-9
		Navigating datasets.	18-9
		Using the First and Last methods	18-10
		Using the Next and Prior methods	18-10
		Using the MoveBy method.	18-10
		Using the Eof and Bof properties	18-11
		Eof	18-11
		Bof	18-12
		Marking and returning to records	18-12
		Searching datasets	18-14

Using Locate	18-14	Defining a lookup field	19-9
Using Lookup.	18-15	Defining an aggregate field	19-11
Displaying and editing a subset of data		Deleting persistent field components	19-11
using filters	18-16	Setting persistent field properties	
Enabling and disabling filtering	18-16	and events	19-12
Creating filters	18-16	Setting display and edit properties at	
Setting the Filter property.	18-17	design time.	19-12
Writing an OnFilterRecord event		Setting field component properties	
handler.	18-18	at runtime	19-13
Switching filter event handlers		Creating attribute sets for field	
at runtime	18-18	components	19-14
Setting filter options	18-19	Associating attribute sets with field	
Navigating records in a filtered		components	19-14
dataset	18-19	Removing attribute associations.	19-15
Modifying data.	18-20	Controlling and masking user input	19-15
Editing records	18-20	Using default formatting for numeric,	
Adding new records	18-21	date, and time fields.	19-16
Inserting records	18-22	Handling events	19-16
Appending records	18-22	Working with field component methods	
Deleting records	18-22	at runtime	19-17
Posting data to the database	18-23	Displaying, converting, and accessing field	
Canceling changes	18-23	values.	19-18
Modifying entire records	18-23	Displaying field component values in	
Using dataset events.	18-25	standard controls	19-18
Aborting a method	18-25	Converting field values.	19-18
Using OnCalcFields	18-26	Accessing field values with the default	
Using BDE-enabled datasets	18-26	dataset property	19-20
Overview of BDE-enablement	18-27	Accessing field values with a dataset's	
Handling database and session		Fields property.	19-20
connections	18-28	Accessing field values with a dataset's	
Using the DatabaseName and		FieldByName method.	19-20
SessionName properties	18-28	Checking a field's current value.	19-21
Working with BDE handle		Setting a default value for a field	19-21
properties	18-29	Working with constraints	19-21
Using cached updates	18-29	Creating a custom constraint.	19-22
Caching BLOBs.	18-30	Using server constraints	19-22
Chapter 19		Using object fields	19-23
Working with field components	19-1	Displaying ADT and array fields	19-23
Understanding field components	19-2	Working with ADT fields.	19-24
Dynamic field components	19-3	Accessing ADT field values.	19-24
Persistent field components.	19-4	Working with array fields	19-25
Creating persistent fields	19-5	Accessing array field values	19-25
Arranging persistent fields	19-6	Working with dataset fields	19-26
Defining new persistent fields	19-6	Displaying dataset fields	19-26
Defining a data field	19-7	Accessing data in a nested dataset.	19-27
Defining a calculated field.	19-8	Working with reference fields	19-27
Programming a calculated field	19-9	Displaying reference fields	19-27
		Accessing data in a reference field	19-27

Chapter 20

Working with tables **20-1**

Using table components.	20-1
Setting up a table component.	20-2
Specifying a database location	20-2
Specifying a table name	20-3
Specifying the table type for local tables.	20-3
Opening and closing a table.	20-4
Controlling read/write access to a table.	20-4
Searching for records	20-5
Searching for records based on indexed fields	20-6
Executing a search with Goto methods	20-6
Executing a search with Find methods	20-7
Specifying the current record after a successful search	20-7
Searching on partial keys	20-8
Searching on alternate indexes	20-8
Repeating or extending a search	20-8
Sorting records	20-9
Retrieving a list of available indexes with GetIndexNames.	20-9
Specifying an alternative index with IndexName	20-9
Specifying a dBASE index file.	20-9
Specifying sort order for SQL tables	20-10
Specifying fields with Index FieldNames	20-10
Examining the field list for an index	20-10
Working with a subset of data	20-11
Understanding the differences between ranges and filters	20-11
Creating and applying a new range	20-12
Setting the beginning of a range	20-12
Setting the end of a range	20-13
Setting start- and end-range values.	20-13
Specifying a range based on partial keys.	20-14
Including or excluding records that match boundary values	20-14
Applying a range	20-14
Canceling a range.	20-15
Modifying a range	20-15
Editing the start of a range.	20-15
Editing the end of a range	20-15
Deleting all records in a table.	20-16
Deleting a table.	20-16

Renaming a table	20-16
Creating a table	20-17
Importing data from another table	20-18
Using TBatchMove	20-19
Creating a batch move component	20-20
Specifying a batch move mode	20-21
Appending	20-21
Updating	20-21
Appending and updating.	20-21
Copying.	20-21
Deleting.	20-22
Mapping data types.	20-22
Executing a batch move.	20-23
Handling batch move errors	20-23
Synchronizing tables linked to the same database table	20-24
Creating master/detail forms	20-24
Building an example master/detail form	20-25
Working with nested tables	20-26
Setting up a nested table component	20-26

Chapter 21

Working with queries **21-1**

Using queries effectively.	21-1
Queries for desktop developers	21-2
Queries for server developers	21-3
What databases can you access with a query component?	21-4
Using a query component: an overview	21-4
Specifying the SQL statement to execute.	21-5
Specifying the SQL property at design time	21-6
Specifying an SQL statement at runtime	21-7
Setting the SQL property directly	21-7
Loading the SQL property from a file	21-8
Loading the SQL property from string list object	21-8
Setting parameters	21-8
Supplying parameters at design time.	21-9
Supplying parameters at runtime	21-10
Using a data source to bind parameters	21-10
Executing a query.	21-12
Executing a query at design time	21-12
Executing a query at runtime	21-12
Executing a query that returns a result set.	21-13
Executing a query without a result set	21-13

Preparing a query	21-13
Unpreparing a query to release resources. . .	21-14
Creating heterogeneous queries	21-14
Improving query performance	21-15
Disabling bi-directional cursors.	21-15
Working with result sets	21-16
Enabling editing of a result set	21-16
Local SQL requirements for a live result set	21-16
Restrictions on live queries	21-16
Remote server SQL requirements for a live result set	21-17
Restrictions on updating a live result set	21-17
Updating a read-only result set.	21-17

Chapter 22 Working with stored procedures 22-1

When should you use stored procedures? . .	22-2
Using a stored procedure	22-2
Creating a stored procedure component. .	22-3
Creating a stored procedure.	22-4
Preparing and executing a stored procedure	22-5
Using stored procedures that return result sets	22-5
Retrieving a result set with a TQuery .	22-5
Retrieving a result set with a TStoredProc	22-6
Using stored procedures that return data using parameters	22-7
Retrieving individual values with a TQuery	22-7
Retrieving individual values with a TStoredProc	22-7
Using stored procedures that perform actions on data	22-8
Executing an action stored procedure with a TQuery	22-8
Executing an action stored procedure with a TStoredProc	22-9
Understanding stored procedure parameters	22-10
Using input parameters	22-10
Using output parameters	22-11
Using input/output parameters	22-12
Using the result parameter	22-12
Accessing parameters at design time. . .	22-13

Setting parameter information at design time	22-13
Creating parameters at runtime	22-14
Binding parameters	22-15
Viewing parameter information at design time.	22-16
Working with Oracle overloaded stored procedures	22-17

Chapter 23 Working with ADO components 23-1

Overview of ADO components	23-1
Connecting to ADO data stores	23-2
Connecting to a data store using TADOConnection	23-3
Using a TADOConnection versus a dataset'sConnectionString	23-3
Specifying the connection.	23-3
Accessing the connection object	23-4
Activating and deactivating the connection	23-4
Determining what a connection component is doing	23-5
Fine-tuning a connection	23-5
Specifying connection attributes	23-5
Controlling timeouts	23-7
Controlling the connection login. . . .	23-7
Listing tables and stored procedures	23-8
Accessing the connection's datasets . . .	23-8
Accessing the connection's commands .	23-9
Listing available tables	23-9
Listing available stored procedures . . .	23-10
Working with (connection) transactions . .	23-11
Using transaction methods	23-11
Using transaction events	23-11
Using ADO datasets	23-11
Features common to all ADO dataset components	23-12
Modifying data.	23-12
Navigating in a dataset	23-12
Using visual data-aware controls	23-13
Connecting to a data store using ADO dataset components.	23-13
Working with record sets	23-14
Using batch updates.	23-14
Loading data from and saving data to files	23-16
Using parameters in commands	23-17
Using TADODataset	23-18

Retrieving a dataset using a command	23-19
Using TADOTable	23-19
Specifying the table to use	23-20
Using TADOQuery.	23-20
Specifying SQL statements	23-21
Executing SQL statements	23-21
Using TADOStoredProc	23-22
Specifying the stored procedure	23-22
Executing the stored procedure	23-23
Using parameters with stored procedures	23-24
Executing commands	23-26
Specifying the command	23-27
Using the Execute method.	23-27
Canceling commands	23-27
Retrieving result sets with commands.	23-28
Handling command parameters	23-28

Chapter 24

Creating and using a client dataset 24-1

Working with data using a client dataset	24-2
Navigating data in client datasets	24-2
Limiting what records appear.	24-2
Representing master/detail relationships.	24-3
Constraining data values	24-3
Making data read-only.	24-4
Editing data.	24-4
Undoing changes	24-5
Saving changes	24-5
Sorting and indexing.	24-6
Adding a new index	24-6
Deleting and switching indexes.	24-7
Using indexes to group data.	24-7
Indexing on the fly	24-8
Representing calculated values	24-8
Using internally calculated fields in client datasets	24-9
Using maintained aggregates	24-9
Specifying aggregates	24-10
Aggregating over groups of records	24-11
Obtaining aggregate values	24-11
Adding application-specific information to the data	24-12
Copying data from another dataset	24-12
Assigning data directly	24-12
Cloning a client dataset cursor	24-13
Using a client dataset with a data provider.	24-14

Specifying a data provider	24-14
Getting parameters from the application server	24-15
Passing parameters to the application server	24-15
Sending query or stored procedure parameters	24-16
Limiting records with parameters	24-16
Overriding the dataset on the application server	24-17
Requesting data from an application server	24-17
Handling constraints	24-18
Handling constraints from the server	24-19
Adding custom constraints	24-19
Updating records	24-20
Applying updates	24-20
Reconciling update errors.	24-21
Refreshing records.	24-22
Communicating with providers using custom events	24-23
Using a client dataset with flat-file data	24-24
Creating a new dataset	24-24
Loading data from a file or stream	24-24
Merging changes into data	24-25
Saving data to a file or stream	24-25

Chapter 25

Working with cached updates 25-1

Deciding when to use cached updates	25-1
Using cached updates	25-2
Enabling and disabling cached updates	25-3
Fetching records	25-3
Applying cached updates	25-4
Applying cached updates with a database component method	25-5
Applying cached updates with dataset component methods.	25-6
Applying updates for master/detail tables	25-6
Canceling pending cached updates	25-7
Cancelling pending updates and disabling further cached updates	25-8
Canceling pending cached updates	25-8
Canceling updates to the current record	25-8
Undeleting cached records	25-8
Specifying visible records in the cache	25-9
Checking update status.	25-10
Using update objects to update a dataset	25-11

Specifying the UpdateObject property for a dataset	25-11	Enabling mouse, keyboard, and timer events	26-5
Using a single update object.	25-12	Using data sources	26-5
Using multiple update objects.	25-12	Using TDataSource properties	26-6
Creating SQL statements for update components	25-13	Setting the DataSet property	26-6
Creating SQL statements at design time	25-13	Setting the Name property	26-6
Understanding parameter substitution in update SQL statements	25-14	Setting the Enabled property	26-7
Composing update SQL statements	25-15	Setting the AutoEdit property	26-7
Using an update component's Query property	25-16	Using TDataSource events	26-7
Using the DeleteSQL, InsertSQL, and ModifySQL properties	25-17	Using the OnDataChange event	26-7
Executing update statements	25-18	Using the OnUpdateData event	26-7
Calling the Apply method	25-18	Using the OnStateChange event	26-7
Calling the SetParams method	25-18	Controls that represent a single field	26-8
Calling the ExecSQL method	25-19	Displaying data as labels	26-8
Using dataset components to update a dataset	25-20	Displaying and editing fields in an edit box	26-9
Updating a read-only result set	25-21	Displaying and editing text in a memo control.	26-9
Controlling the update process.	25-21	Displaying and editing text in a rich edit memo control	26-10
Determining if you need to control the updating process	25-22	Displaying and editing graphics fields in an image control	26-10
Creating an OnUpdateRecord event handler.	25-22	Displaying and editing data in list and combo boxes	26-11
Handling cached update errors	25-23	Displaying and editing data in a list box.	26-11
Referencing the dataset to which to apply updates	25-24	Displaying and editing data in a combo box	26-12
Indicating the type of update that generated an error	25-24	Displaying and editing data in lookup list and combo boxes	26-12
Specifying the action to take	25-25	Specifying a list based on a lookup field	26-13
Working with error message text	25-25	Specifying a list based on a secondary data source	26-13
Accessing a field's OldValue, NewValue, and CurValue properties.	25-26	Setting lookup list and combo box properties	26-14
		Searching incrementally for list item values	26-14
Chapter 26		Handling Boolean field values with check boxes.	26-14
Using data controls	26-1	Restricting field values with radio controls	26-15
Using common data control features	26-1	Viewing and editing data with TDBGrid.	26-16
Associating a data control with a dataset	26-2	Using a grid control in its default state	26-17
Editing and updating data	26-3	Creating a customized grid.	26-17
Enabling editing in controls on user entry	26-3	Understanding persistent columns	26-18
Editing data in a control	26-3	Determining the source of a column property at runtime	26-18
Disabling and enabling data display.	26-4	Creating persistent columns	26-19
Refreshing data display	26-5		

Deleting persistent columns	26-20	Using the Decision Cube editor	27-7
Arranging the order of persistent columns	26-20	Viewing and changing dimension settings	27-8
Defining a lookup list column	26-20	Setting the maximum available dimensions and summaries	27-8
Defining a pick list column	26-20	Viewing and changing design options	27-8
Putting a button in a column	26-21	Using decision sources	27-9
Setting column properties at design time	26-21	Properties and events	27-9
Restoring default values to a column	26-22	Using decision pivots	27-9
Displaying ADT and array fields	26-22	Decision pivot properties	27-10
Setting grid options	26-24	Creating and using decision grids	27-10
Editing in the grid	26-25	Creating decision grids	27-10
Rearranging column order at design time	26-26	Using decision grids	27-11
Rearranging column order at runtime	26-26	Opening and closing decision grid fields	27-11
Controlling grid drawing	26-26	Reorganizing rows and columns in decision grids	27-11
Responding to user actions at runtime	26-27	Drilling down for detail in decision grids	27-11
Creating a grid that contains other data-aware controls	26-27	Limiting dimension selection in decision grids	27-12
Navigating and manipulating records	26-29	Decision grid properties	27-12
Choosing navigator buttons to display	26-30	Creating and using decision graphs	27-13
Hiding and showing navigator buttons at design time	26-30	Creating decision graphs	27-13
Hiding and showing navigator buttons at runtime	26-30	Using decision graphs	27-13
Displaying fly-over help	26-31	The decision graph display	27-15
Using a single navigator for multiple datasets	26-31	Customizing decision graphs	27-15
		Setting decision graph template defaults	27-16
		Customizing decision graph series	27-17
		Decision support components at runtime	27-18
		Decision pivots at runtime	27-18
		Decision grids at runtime	27-18
		Decision graphs at runtime	27-19
		Decision support components and memory control	27-19
		Setting maximum dimensions, summaries, and cells	27-19
		Setting dimension state	27-20
		Using paged dimensions	27-20
Chapter 27			
Using decision support components	27-1		
Overview	27-1		
About crosstabs	27-2		
One-dimensional crosstabs	27-2		
Multidimensional crosstabs	27-3		
Guidelines for using decision support components	27-3		
Using datasets with decision support components	27-4		
Creating decision datasets with TQuery or TTable	27-5		
Creating decision datasets with the Decision Query editor	27-6		
Using the Decision Query editor	27-6		
Decision query properties	27-7		
Using decision cubes	27-7		
Decision cube properties and events	27-7		
		Part III	
		Writing distributed applications	
		<hr/>	
		Chapter 28	
		Writing CORBA applications	28-1
		Overview of a CORBA application	28-2
		Understanding stubs and skeletons	28-2
		Using Smart Agents	28-3

Activating server applications	28-3
Binding interface calls dynamically	28-4
Writing CORBA servers	28-4
Using the CORBA wizards	28-4
Defining object interfaces	28-5
Automatically generated code	28-7
Registering server interfaces	28-8
Registering interfaces with the Interface Repository	28-8
Registering interfaces with the Object Activation Daemon	28-9
Writing CORBA clients	28-11
Using stubs	28-12
Using the dynamic invocation interface	28-12
Obtaining the interface	28-13
Calling interfaces with DII	28-13
Customizing CORBA applications	28-14
Displaying objects in the user interface	28-15
Exposing and hiding CORBA objects	28-15
Passing client information to server objects	28-16
Deploying CORBA applications	28-16
Configuring Smart Agents	28-17
Starting the Smart Agent	28-17
Configuring ORB domains	28-18
Connecting Smart Agents on different local networks	28-18

Chapter 29

Creating Internet server applications

29-1

Terminology and standards	29-1
Parts of a Uniform Resource Locator	29-2
URI vs. URL	29-2
HTTP request header information	29-2
HTTP server activity	29-3
Composing client requests	29-3
Serving client requests	29-4
Responding to client requests	29-4
Web server applications	29-5
Types of Web server applications	29-5
ISAPI and NSAPI	29-5
CGI stand-alone	29-5
Win-CGI stand-alone	29-5
Creating Web server applications	29-6
The Web module	29-6
The Web Application object	29-7
The structure of a Web server application	29-7
The Web dispatcher	29-8

Adding actions to the dispatcher	29-9
Dispatching request messages	29-9
Action items	29-10
Determining when action items fire	29-10
The target URL	29-10
The request method type	29-10
Enabling and disabling action items	29-11
Choosing a default action item	29-11
Responding to request messages with action items	29-12
Sending the response	29-12
Using multiple action items	29-12
Accessing client request information	29-13
Properties that contain request header information	29-13
Properties that identify the target	29-13
Properties that describe the Web client	29-13
Properties that identify the purpose of the request	29-14
Properties that describe the expected response	29-14
Properties that describe the content	29-14
The content of HTTP request messages	29-15
Creating HTTP response messages	29-15
Filling in the response header	29-15
Indicating the response status	29-15
Indicating the need for client action	29-16
Describing the server application	29-16
Describing the content	29-16
Setting the response content	29-16
Sending the response	29-17
Generating the content of response messages	29-17
Using page producer components	29-17
HTML templates	29-18
Specifying the HTML template	29-19
Converting HTML-transparent tags	29-19
Using page producers from an action item	29-19
Chaining page producers together	29-20
Using database information in responses	29-21
Adding a session to the Web module	29-21
Representing database information in HTML	29-22
Using dataset page producers	29-22
Using table producers	29-23
Specifying the table attributes	29-23

Specifying the row attributes	29-23
Specifying the columns.	29-23
Embedding tables in HTML documents	29-24
Setting up a dataset table producer.	29-24
Setting up a query table producer	29-24
Debugging server applications.	29-25
Debugging ISAPI and NSAPI applications	29-25
Debugging under Windows NT.	29-25
Debugging with a Microsoft IIS server.	29-25
Debugging under MTS.	29-26
Debugging with a Windows 95 Personal Web Server	29-27
Debugging with Netscape Server Version 2.0	29-27
Debugging CGI and Win-CGI applications	29-28
Simulating the server.	29-28
Debugging as a DLL	29-28

Getting information about connections	30-7
Closing server connections	30-8
Responding to socket events.	30-8
Error events	30-8
Client events.	30-8
Server events.	30-9
Events when listening.	30-9
Events with client connections	30-9
Reading and writing over socket connections	30-10
Non-blocking connections	30-10
Reading and writing events	30-10
Blocking connections	30-11
Using threads with blocking connections	30-11
Using TWinSocketStream.	30-12
Writing client threads	30-12
Writing server threads.	30-13

Chapter 30

Working with sockets **30-1**

Implementing services	30-1
Understanding service protocols	30-2
Communicating with applications	30-2
Services and ports	30-2
Types of socket connections.	30-2
Client connections	30-3
Listening connections	30-3
Server connections	30-3
Describing sockets	30-3
Describing the host.	30-4
Choosing between a host name and an IP address	30-4
Using ports	30-5
Using socket components.	30-5
Using client sockets	30-5
Specifying the desired server	30-6
Forming the connection	30-6
Getting information about the connection	30-6
Closing the connection.	30-6
Using server sockets	30-6
Specifying the port	30-7
Listening for client requests	30-7
Connecting to clients	30-7

Part IV Creating custom components

Chapter 31

Overview of component creation **31-1**

Visual Component Library.	31-1
Components and classes	31-2
How do you create components?	31-2
Modifying existing controls	31-3
Creating windowed controls.	31-3
Creating graphic controls.	31-4
Subclassing Windows controls.	31-4
Creating nonvisual components	31-4
What goes into a component?	31-5
Removing dependencies	31-5
Properties, methods, and events.	31-5
Properties	31-6
Events	31-6
Methods.	31-6
Graphics encapsulation.	31-7
Registration	31-7
Creating a new component	31-7
Using the Component wizard	31-8
Creating a component manually.	31-10
Creating a unit file.	31-10
Deriving the component	31-11
Registering the component	31-11
Testing uninstalled components.	31-12

Chapter 32	
Object-oriented programming for component writers	32-1
Defining new classes	32-1
Deriving new classes.	32-2
To change class defaults to avoid repetition.	32-2
To add new capabilities to a class.	32-2
Declaring a new component class	32-3
Ancestors, descendants, and class hierarchies	32-3
Controlling access	32-4
Hiding implementation details	32-4
Defining the component writer's interface	32-5
Defining the runtime interface	32-6
Defining the design-time interface	32-6
Dispatching methods	32-7
Static methods	32-7
Virtual methods	32-8
Overriding methods	32-8
Abstract class members	32-9
Classes and pointers	32-9

Chapter 33	
Creating properties	33-1
Why create properties?	33-1
Types of properties.	33-2
Publishing inherited properties	33-2
Defining properties	33-3
The property declaration	33-3
Internal data storage	33-4
Direct access.	33-4
Access methods.	33-5
The read method	33-6
The write method	33-6
Default property values	33-7
Specifying no default value	33-7
Creating array properties	33-8
Storing and loading properties.	33-8
Using the store-and-load mechanism	33-9
Specifying default values	33-9
Determining what to store.	33-10
Initializing after loading.	33-10
Storing and loading unpublished properties	33-11
Creating methods to store and load property values	33-11
Overriding the DefineProperties method.	33-12

Chapter 34	
Creating events	34-1
What are events?	34-1
Events are method pointers	34-2
Events are properties	34-2
Event types are method-pointer types	34-3
Event-handler types are procedures.	34-3
Event handlers are optional	34-4
Implementing the standard events	34-4
Identifying standard events	34-4
Standard events for all controls	34-5
Standard events for standard controls	34-5
Making events visible.	34-5
Changing the standard event handling.	34-6
Defining your own events	34-6
Triggering the event.	34-6
Two kinds of events	34-7
Defining the handler type	34-7
Simple notifications	34-7
Event-specific handlers	34-7
Returning information from the handler	34-8
Declaring the event	34-8
Event names start with "On".	34-8
Calling the event.	34-8

Chapter 35	
Creating methods	35-1
Avoiding dependencies	35-1
Naming methods	35-2
Protecting methods	35-2
Methods that should be public.	35-3
Methods that should be protected.	35-3
Abstract methods	35-3
Making methods virtual	35-4
Declaring methods	35-4

Chapter 36	
Using graphics in components	36-1
Overview of graphics.	36-1
Using the canvas	36-3
Working with pictures	36-3
Using a picture, graphic, or canvas	36-3
Loading and storing graphics	36-4
Handling palettes	36-4
Specifying a palette for a control.	36-5
Off-screen bitmaps	36-5
Creating and managing off-screen bitmaps	36-6

Copying bitmapped images.	36-6
Responding to changes	36-7

Chapter 37 Handling messages 37-1

Understanding the message-handling system	37-1
What's in a Windows message?	37-2
Dispatching messages	37-2
Tracing the flow of messages	37-3
Changing message handling	37-3
Overriding the handler method	37-3
Using message parameters	37-4
Trapping messages	37-4
Creating new message handlers	37-5
Defining your own messages	37-5
Declaring a message identifier	37-5
Declaring a message-record type	37-6
Declaring a new message-handling method.	37-6

Chapter 38 Making components available at design time 38-1

Registering components.	38-1
Declaring the Register procedure.	38-2
Writing the Register procedure	38-2
Specifying the components	38-2
Specifying the palette page	38-3
Using the RegisterComponents function	38-3
Adding palette bitmaps	38-3
Providing Help for your component.	38-4
Creating the Help file	38-4
Creating the entries.	38-4
Making component help context-sensitive	38-6
Adding component help files	38-6
Adding property editors	38-6
Deriving a property-editor class	38-7
Editing the property as text	38-8
Displaying the property value.	38-8
Setting the property value	38-8
Editing the property as a whole	38-9
Specifying editor attributes	38-10
Registering the property editor	38-11
Adding component editors	38-12
Adding items to the context menu	38-12
Specifying menu items	38-12

Implementing commands.	38-13
Changing the double-click behavior	38-13
Adding clipboard formats	38-14
Registering the component editor	38-15
Property categories	38-15
Registering one property at a time	38-16
Registering multiple properties at once.	38-16
Property category classes.	38-17
Built-in property categories.	38-17
Deriving new property categories	38-18
Using the IsPropertyInCategory function.	38-18
Compiling components into packages	38-19
Troubleshooting custom components.	38-19

Chapter 39 Modifying an existing component 39-1

Creating and registering the component	39-1
Modifying the component class	39-2
Overriding the constructor.	39-2
Specifying the new default property value	39-3

Chapter 40 Creating a graphic component 40-1

Creating and registering the component	40-1
Publishing inherited properties	40-2
Adding graphic capabilities	40-2
Determining what to draw	40-3
Declaring the property type	40-3
Declaring the property	40-3
Writing the implementation method	40-4
Overriding the constructor and Sdestructor	40-4
Changing default property values.	40-4
Publishing the pen and brush	40-5
Declaring the class fields	40-5
Declaring the access properties.	40-6
Initializing owned classes.	40-6
Setting owned classes' properties	40-7
Drawing the component image	40-8
Refining the shape drawing	40-9

Chapter 41 Customizing a grid 41-1

Creating and registering the component	41-1
Publishing inherited properties	41-2
Changing initial values.	41-3
Resizing the cells	41-4

Filling in the cells	41-4
Tracking the date	41-5
Storing the internal date	41-5
Accessing the day, month, and year	41-6
Generating the day numbers	41-7
Selecting the current day.	41-9
Navigating months and years	41-9
Navigating days	41-10
Moving the selection.	41-10
Providing an OnChange event	41-11
Excluding blank cells	41-12

Chapter 42

Making a control data aware 42-1

Creating a data-browsing control	42-1
Creating and registering the component.	42-2
Making the control read-only.	42-2
Adding the ReadOnly property.	42-3
Allowing needed updates	42-3
Adding the data link	42-4
Declaring the class field	42-4
Declaring the access properties	42-5
An example of declaring access properties	42-5
Initializing the data link	42-6
Responding to data changes	42-6
Creating a data-editing control.	42-7
Changing the default value of FReadOnly	42-8
Handling mouse-down and key-down messages.	42-8
Responding to mouse-down messages	42-8
Responding to key-down messages	42-9
Updating the field data-link class	42-10
Modifying the Change method	42-10
Updating the dataset.	42-11

Chapter 43

Making a dialog box a component 43-1

Defining the component interface	43-1
Creating and registering the component	43-2
Creating the component interface	43-3
Including the form unit	43-3
Adding interface properties.	43-3
Adding the Execute method	43-4
Testing the component	43-6

Part V

Developing COM-based applications

Chapter 44

Overview of COM technologies 44-1

COM as a specification and implementation	44-1
COM extensions	44-2
Parts of a COM application	44-2
COM interfaces	44-3
The fundamental COM interface, IUnknown	44-4
COM interface pointers	44-4
COM servers	44-5
CoClasses and class factories	44-5
In-process, out-of-process, and remote servers.	44-6
The marshaling mechanism	44-7
COM clients	44-8
COM extensions.	44-8
Automation servers and controllers.	44-11
ActiveX controls	44-11
Type libraries.	44-12
The content of type libraries	44-12
Creating type libraries.	44-12
When to use type libraries	44-13
Accessing type libraries	44-13
Benefits of using type libraries	44-14
Using type library tools	44-14
Active Server Pages	44-14
Active Documents.	44-15
Visual cross-process objects	44-15
Implementing COM objects with wizards.	44-16

Chapter 45

Creating a simple COM object 45-1

Overview of creating a COM object.	45-1
Designing a COM object	45-2
Creating a COM object with the COM object wizard	45-2
COM object instancing types	45-3
Choosing a threading model.	45-3
Writing an object that supports the free threading model	45-5
Writing an object that supports the apartment threading model	45-5

Registering a COM object	45-6
Testing a COM object	45-6

Chapter 46

Creating an Automation controller 46-1

Creating an Automation controller by importing a type library	46-1
Handling events in an automation controller	46-2
Connecting to and disconnecting from a server	46-3
Controlling an Automation server using a dual interface	46-3
Controlling an Automation server using a dispatch interface	46-4
Example: Printing a document with Microsoft Word	46-4
Getting more information	46-7

Chapter 47

Creating an Automation server 47-1

Creating an Automation object for an application	47-1
Managing events in your Automation object	47-2
Exposing an application's properties, methods, and events	47-3
Exposing a property for Automation	47-3
Exposing a method for Automation	47-3
Exposing an event for Automation	47-4
Getting more information	47-5
Registering an application as an Automation server	47-5
Registering an in-process server	47-5
Registering an out-of-process server	47-5
Testing and debugging the application	47-6
Automation interfaces	47-6
Dual interfaces	47-6
Dispatch interfaces	47-7
Custom interfaces	47-8
Marshaling data	47-8
Automation compatible types	47-9
Type restrictions for automatic marshaling	47-9
Custom marshaling	47-10

Chapter 48

Creating an ActiveX control 48-1

Overview of ActiveX control creation	48-1
--	------

Elements of an ActiveX control	48-2
VCL control	48-2
Type library	48-2
Properties, methods, and events	48-3
Property page	48-3
Designing an ActiveX control	48-3
Generating an ActiveX control from a VCL control	48-4
Licensing ActiveX controls	48-5
Generating an ActiveX control based on a VCL form	48-6
Working with properties, methods, and events in an ActiveX control	48-8
Adding additional properties, methods, and events	48-9
How Delphi adds properties	48-9
How Delphi adds methods	48-10
How Delphi adds events	48-10
Enabling simple data binding with the type library	48-11
Enabling simple data binding of ActiveX controls in the Delphi container	48-12
Creating a property page for an ActiveX control	48-13
Creating a new property page	48-14
Adding controls to a property page	48-14
Associating property page controls with ActiveX control properties	48-14
Updating the property page	48-14
Updating the object	48-15
Connecting a property page to an ActiveX control	48-15
Exposing properties of an ActiveX control	48-15
Registering an ActiveX control	48-17
Testing an ActiveX control	48-17
Deploying an ActiveX control on the Web	48-17
Setting options	48-18
Web Deploy Options Default checkbox	48-19
INF file	48-19
Option combinations	48-20
Project tab	48-20
Packages tab	48-21
Packages used by this project	48-22
CAB options	48-22
Output options	48-22
Directory and URL options	48-22

Additional Files tab	48-22	CoClass pages	50-16
Files associated with project	48-22	Attributes page for a CoClass	50-16
CAB options	48-23	CoClass Implements page	50-16
Output options	48-23	CoClass flags	50-17
Directory and URL options	48-23	Enumeration type information	50-18
Code Signing tab	48-23	Attributes page for an enum	50-18
Required information	48-24	Enumeration members	50-18
Optional information	48-24	Alias type information	50-19
Timestamp server	48-24	Attributes page for an alias	50-19
Cryptographic digest algorithm	48-25	Record type information	50-19
		Attributes page for a record	50-20
		Record members	50-20
		Union type information	50-20
		Attributes page for a union	50-21
		Union members	50-21
		Module type information	50-21
		Attributes page for a module	50-22
		Module members	50-22
		Module methods	50-22
		Module constants	50-23
		Creating new type libraries	50-23
		Valid types	50-23
		SafeArrays	50-25
		Using Object Pascal or IDL syntax	50-25
		Attribute specifications	50-25
		Interface syntax	50-27
		Dispatch interface syntax	50-28
		CoClass syntax	50-28
		Enum syntax	50-29
		Alias syntax	50-29
		Record syntax	50-29
		Union syntax	50-30
		Module syntax	50-30
		Creating a new type library	50-31
		Opening an existing type library	50-31
		Adding an interface to the type	
		library	50-32
		Adding properties and methods to an	
		interface or dispinterface	50-32
		Adding a CoClass to the type library	50-33
		Adding an enumeration to the type	
		library	50-33
		Saving and registering type library	
		information	50-33
		Apply Updates dialog	50-34
		Saving a type library	50-35
		Refreshing the type library	50-35
		Registering the type library	50-35
		Exporting an IDL file	50-35
		Deploying type libraries	50-36
Chapter 49			
Creating an Active Server Page	49-1		
Creating an Active Server Page object	49-2		
Creating ASPs for in-process or			
out-of-process servers	49-3		
Registering an application as an			
Active Server Page object	49-4		
Registering an in-process server	49-4		
Registering an out-of-process server	49-4		
Testing and debugging the Active Server			
Page application	49-4		
Chapter 50			
Working with type libraries	50-1		
Type Library editor	50-2		
Toolbar	50-3		
Object list pane	50-5		
Status bar	50-5		
Pages of type information	50-6		
Attributes page	50-6		
Text page	50-7		
Flags page	50-7		
Type library information	50-7		
Attributes page for a type library	50-8		
Uses page for a type library	50-8		
Flags page for a type library	50-8		
Interface pages	50-9		
Attributes page for an interface	50-9		
Interface flags	50-10		
Interface members	50-10		
Interface methods	50-10		
Interface properties	50-11		
Property and method parameters			
page	50-12		
Dispatch type information	50-14		
Attributes page for dispatch	50-15		
Dispatch flags page	50-15		
Dispatch members	50-15		

Chapter 51

Creating MTS objects **51-1**

Microsoft Transaction Server components . . .	51-2
Requirements for an MTS component . . .	51-4
Managing resources with just-in-time activation and resource pooling	51-4
Just-in-time activation	51-4
Resource pooling	51-5
Releasing resources	51-6
Object pooling	51-6
Accessing the object context.	51-6
MTS transaction support	51-7
Transaction attributes	51-7
Object context holds transaction attribute	51-8
Stateful and stateless objects	51-8
Enabling multiple objects to support transactions	51-9
MTS or client-controlled transactions . . .	51-10
Advantage of transactions.	51-10
Transaction timeout	51-10
Role-based security	51-11
Resource dispensers	51-11
BDE resource dispenser	51-12
Shared property manager	51-12

Example: Sharing properties among MTS object instances.	51-12
Tips for using the Shared Property Manager.	51-13
Base clients and MTS components	51-14
MTS underlying technologies, COM and DCOM.	51-14
Overview of creating MTS objects	51-15
Using the MTS Object wizard	51-15
Choosing a threading model for an MTS object	51-16
MTS activities	51-17
Setting the transaction attribute.	51-17
Passing object references	51-18
Using the SafeRef method	51-18
Callbacks	51-19
Setting up a transaction object on the client side	51-19
Setting up a transaction object on the server side	51-20
Debugging and testing MTS objects	51-20
Installing MTS objects into an MTS package.	51-21
Administering MTS objects with the MTS Explorer	51-22
Using MTS documentation	51-22

Index

I-1

Tables

1.1	Typefaces and symbols	1-2	15.1	Provider options	15-3
2.1	Component palette pages	2-10	15.2	UpdateStatus values	15-7
3.1	RTL exceptions	3-6	15.3	UpdateMode values	15-8
3.2	Object Pascal character types	3-23	15.4	ProviderFlags values	15-8
3.3	String comparison routines	3-27	16.1	Database-related informational methods for session components	16-9
3.4	Case conversion routines	3-28	16.2	TSessionList properties and methods	16-16
3.5	String modification routines	3-28	18.1	Values for the dataset State property	18-3
3.6	Sub-string routines	3-28	18.2	Navigational methods of datasets	18-9
3.7	Attribute constants and values	3-34	18.3	Navigational properties of datasets.	18-9
5.1	Sample captions and their derived names	5-18	18.4	Comparison and logical operators that can appear in a filter	18-17
5.2	Menu Designer context menu commands	5-23	18.5	FilterOptions values	18-19
5.3	Setting speed buttons' appearance.	5-30	18.6	Filtered dataset navigational methods	18-19
5.4	Setting tool buttons' appearance.	5-32	18.7	Dataset methods for inserting, updating, and deleting data	18-20
5.5	Setting a cool button's appearance.	5-33	18.8	Methods that work with entire records	18-24
6.1	Properties of selected text.	6-8	18.9	Dataset events.	18-25
6.2	Fixed vs. variable owner-draw styles	6-12	18.10	TDBDataSet database and session properties and function	18-28
7.1	Graphic object types.	7-3	18.11	Properties, events, and methods for cached updates	18-29
7.2	Common properties of the Canvas object	7-3	19.1	Field components.	19-1
7.3	Common methods of the Canvas object	7-4	19.2	TFloatField properties that affect data display	19-2
7.4	Mouse-event parameters	7-23	19.3	Special persistent field kinds	19-6
7.5	Multimedia device types and their functions	7-31	19.4	Field component properties	19-12
8.1	Thread priorities	8-3	19.5	Field component formatting routines	19-16
8.2	WaitFor return values	8-9	19.6	Field component events	19-16
9.1	Compiled package files	9-2	19.7	Selected field component methods	19-17
9.2	Runtime packages	9-4	19.8	Field component conversion functions.	19-19
9.3	Design-time packages	9-5	19.9	Special conversion results	19-19
9.4	Package-specific compiler directives	9-11	19.10	Types of object field components	19-23
9.5	Package-specific command-line compiler switches	9-12	19.11	Common object field descendant properties	19-23
9.6	Compiled package files	9-13	20.1	Table types recognized by the BDE based on file extension.	20-4
10.1	VCL objects that support BiDi	10-4	20.2	TableType values	20-4
10.2	Estimating string lengths	10-8	20.3	Legacy TTable search methods	20-6
11.1	Application files	11-2	20.4	BatchMove import modes	20-19
11.2	SQL database client software files	11-5	20.5	Batch move modes	20-21
12.1	Data Dictionary interface	12-5			
13.1	Possible values for the TransIsolation property.	13-7			
13.2	Transaction isolation levels.	13-8			
14.1	MIDAS components.	14-3			
14.2	Connection components	14-4			
14.3	AppServer interface members	14-8			
14.4	The javascript libraries	14-32			

23.1	ADO components	23-2	36.1	Canvas capability summary	36-3
23.2	Parameter direction property.	23-24	36.2	Image-copying methods	36-7
24.1	Summary operators for maintained aggregates	24-10	38.1	Predefined property-editor types	38-7
24.2	Client datasets properties and method for handling data requests	24-17	38.2	Methods for reading and writing property values	38-8
25.1	TUpdateRecordType values	25-9	38.3	Property-editor attribute flags.	38-10
25.2	Return values for UpdateStatus	25-10	38.4	Property categories	38-17
25.3	UpdateKind values	25-24	44.1	COM object requirements	44-10
25.4	UpdateAction values	25-25	44.2	Delphi wizards for implementing COM, Automation, and ActiveX objects.	44-17
26.1	Data controls	26-2	45.1	Threading models for COM objects	45-4
26.2	Properties affecting editing in data controls	26-4	50.1	Type library pages	50-6
26.3	Data-aware list box and combo box controls	26-11	50.2	Attributes common to all types	50-7
26.4	TDBLookupListBox and TDBLookupComboBox properties.	26-14	50.3	Type library attributes	50-8
26.5	Column properties.	26-21	50.4	Type library flags	50-8
26.6	Expanded TColumn Title properties.	26-22	50.5	Interface attributes	50-9
26.7	Properties that affect the way ADT and array fields appear in a TDBGrid	26-23	50.6	Interface flags	50-10
26.8	Expanded TDBGrid Options properties.	26-24	50.7	Method attributes.	50-10
26.9	Grid control events	26-27	50.8	Method flags	50-11
26.10	Selected database control grid properties.	26-28	50.9	Property attributes	50-11
26.11	TDBNavigator buttons	26-29	50.10	Property flags	50-12
28.1	Types allowed in a CORBA interface	28-6	50.11	Parameter modifiers (Object Pascal Syntax)	50-13
28.2	irep arguments	28-8	50.12	Parameter flags (IDL syntax)	50-14
28.3	idl2ir arguments	28-9	50.13	Dispinterface attributes	50-15
28.4	OAD arguments	28-9	50.14	CoClass attributes	50-16
28.5	oadutil reg arguments.	28-10	50.15	CoClass Implements page options	50-17
28.6	oadutil unreg arguments	28-11	50.16	CoClass flags	50-17
28.7	ORB methods for creating structured TAny values	28-14	50.17	Enum attributes.	50-18
28.8	CORBA environment variables	28-17	50.18	Alias attributes	50-19
28.9	osagent arguments.	28-17	50.19	Record attributes	50-20
29.1	Web server application components.	29-5	50.20	Union attributes	50-21
29.2	MethodType values	29-11	50.21	Module attributes.	50-22
31.1	Component creation starting points.	31-3	50.22	Module method attributes.	50-22
32.1	Levels of visibility within an object	32-4	50.23	Valid types	50-24
33.1	How properties appear in the Object Inspector	33-2	50.24	Attribute syntax	50-26
			51.1	IObjectContext methods for transaction support.	51-9
			51.2	MTS server objects versus base clients	51-14
			51.3	Threading models for COM objects	51-16
			51.4	Microsoft MTS documentation roadmap	51-22

Figures

2.1	A simple form	2-3	21.1	Sample master/detail query form and data module at design time	21-11
2.2	A simplified hierarchy diagram	2-6	26.1	TDBGrid control	26-16
2.3	Three views of the track bar component	2-14	26.2	TDBGrid control with ObjectView set to False	26-23
2.4	A progress bar	2-21	26.3	TDBGrid control with Expanded set to False	26-23
2.5	A simple data module.	2-34	26.4	TDBGrid control with Expanded set to True	26-24
5.1	A Frame with data-aware controls and a data source component	5-14	26.5	TDBCtrGrid at design time	26-28
5.2	Menu terminology	5-15	26.6	Buttons on the TDBNavigator control	26-29
5.3	MainMenu and PopupMenu components	5-15	27.1	Decision support components at design time	27-2
5.4	Menu Designer for a main menu	5-16	27.2	One-dimensional crosstab	27-3
5.5	Menu Designer for a pop-up menu	5-17	27.3	Three-dimensional crosstab	27-3
5.6	Nested menu structures.	5-20	27.4	Decision graphs bound to different decision sources.	27-14
5.7	Select Menu dialog box	5-23	28.1	The structure of a CORBA application.	28-2
5.8	Sample Insert Template dialog box for menu	5-24	28.2	Separate ORB domains.	28-18
5.9	Save Template dialog box for menus	5-25	28.3	Two Smart Agents on separate local networks.	28-19
5.10	Action list mechanism.	5-36	29.1	Parts of a Uniform Resource Locator	29-2
5.11	Execution cycle for an action	5-38	29.2	Structure of a Server Application	29-8
5.12	Action targets	5-42	31.1	Visual Component Library class hierarchy.	31-2
7.1	Bitmap-dimension dialog box from the BMPDlg unit.	7-20	31.2	Component wizard	31-9
10.1	TListBox set to bdLeftToRight	10-6	44.1	A COM interface	44-3
10.2	TListBox set to bdRightToLeft	10-6	44.2	Interface vtable	44-5
10.3	TListBox set to bdRightToLeftNoAlign	10-6	44.3	In-process server	44-7
10.4	TListBox set to bdRightToLeftReadingOnly	10-6	44.4	Out-of-process and remote servers	44-7
12.1	User-interface to dataset connections in all database applications.	12-7	44.5	COM-based technologies	44-10
12.2	Single-tiered database application architectures	12-8	44.6	Simple COM object interface	44-16
12.3	Two-tiered database application architectures	12-9	44.7	Automation object interface	44-17
12.4	Multi-tiered database architectures	12-10	44.8	ActiveX object interface	44-17
13.1	Components in a BDE-based application	13-3	47.1	Dual interface VTable	47-7
14.1	Web-based multi-tiered database application	14-28	48.1	Mask Edit property page in design mode	48-14
18.1	Delphi Dataset hierarchy	18-1	50.1	Type Library editor	50-2
18.2	Relationship of Inactive and Browse states	18-5	50.2	Object list pane	50-5
18.3	Relationship of Browse to other dataset states	18-6	51.1	MTS object interface	51-2
18.4	Dataset component hierarchy	18-27	51.2	MTS In-process component	51-3
			51.3	An MTS component in an out-of-process server	51-3
			51.4	An MTS component in a remote server process	51-3

Introduction

The *Developer's Guide* describes intermediate and advanced development topics, such as building client/server database applications, writing custom components, and creating Internet Web server applications. It allows you to build applications that meet many industry-standard specifications such as CORBA, TCP/IP, MTS, COM, and ActiveX. The *Developer's Guide* assumes you are familiar with using Delphi and understand fundamental Delphi programming techniques. For an introduction to Delphi programming and the integrated development environment (IDE), see the online Help.

What's in this manual?

This manual contains five parts, as follows:

- **Part I, "Programming with Delphi,"** describes how to build general-purpose Delphi applications. This part provides details on programming techniques you can use in any Delphi application. For example, it describes how to use common Visual Component Library (VCL) objects that make user interface programming easy such as handling strings, manipulating text, implementing the Windows common dialog, toolbars, and cool bars. It also includes chapters on working with graphics, error and exception handling, using DLLs, OLE automation, and writing international applications.

The chapter on deployment details the tasks involved in deploying your application to your application users. For example, it includes information on effective compiler options, using InstallShield Express, licensing issues, and how to determine which packages, DLLs, and other libraries to use when building the production-quality version of your application.

- **Part II, "Developing database applications,"** describes how to build database applications using database tools and components. Delphi lets you access many types of databases. With the forms and reports you create, you can access local databases such as Paradox and dBASE, network SQL server databases like

InterBase and Sybase, and any data source accessible through open database connectivity (ODBC). To implement the more advanced Client/Server database applications, you need the Delphi features available in the Client/Server and Enterprise editions.

- **Part III, “Writing distributed applications,”** describes how to create applications that are distributed over a local area network. These include CORBA applications, and Web server applications such as CGI applications or NSAPI and ISAPI dynamic-link libraries (DLLs). For lower-level support of distributed applications, this section also describes how to work with socket components, that handle the details of communication using TCP/IP and related protocols. The components that support CORBA and Web server applications are available in the Client/Server and Enterprise editions of Delphi. The socket components are available in the Professional version as well.
- **Part IV, “Creating custom components,”** describes how to design and implement your own components, and how to make them available on the Component palette of the IDE. A component can be almost any program element that you want to manipulate at design time. Implementing custom components entails deriving a new class from an existing class type in the VCL class library.
- **Part V, “Developing COM-based applications,”** describes how to build applications that can interoperate with other COM-based API objects on the system such as Win95 Shell extensions or multimedia applications. Delphi contains components that support the ActiveX, COM-based library for COM controls that can be used for general-purpose and Web-based applications. This part also describes how to write servers that can reside in the MTS runtime environment. MTS provides extensive runtime support for security, transactions, and resource pooling.

Support for COM controls is available in all editions of Delphi. To create ActiveX controls, you need the Professional, Client/Server, or Enterprise edition. To create MTS servers, you need the Client/Server or Enterprise edition.

Manual conventions

This manual uses the typefaces and symbols described in Table 1.1 to indicate special text.

Table 1.1 Typefaces and symbols

Typeface or symbol	Meaning
Monospace type	Monospaced text represents text as it appears on screen or in Object Pascal code. It also represents anything you must type.
[]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent Object Pascal keywords or compiler options.

Table 1.1 Typefaces and symbols (continued)

Typeface or symbol	Meaning
<i>Italics</i>	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
<i>Keycaps</i>	This typeface indicates a key on your keyboard. For example, "Press <i>Esc</i> to exit a menu."

Developer support services

Inprise also offers a variety of support options to meet the needs of its diverse developer community. To find out about support offerings for Delphi, refer to <http://www.borland.com/devsupport/delphi>.

Additional Delphi Technical Information documents and answers to Frequently Asked Questions (FAQs) are also available at this Web site.

From the Web site, you can access many newsgroups where Delphi developers exchange information, tips, and techniques. The site also includes a list of books about Delphi.

Ordering printed documentation

For information about ordering additional documentation, refer to the Web site at shop.borland.com.

Programming with Delphi

The chapters in “Programming with Delphi” introduce concepts and skills necessary for creating Delphi applications using any edition of the product. They also introduce the concepts discussed in later sections of the *Developer’s Guide*.

Using Object Pascal with the VCL

This chapter discusses how to use Object Pascal and the object and component library in Delphi applications.

Object Pascal and the VCL

Object Pascal, a set of object-oriented extensions to standard Pascal, is the language of Delphi. The Visual Component Library (VCL) is a hierarchy of classes—written in Object Pascal and tied to the Delphi IDE—that allows you to develop applications quickly. Using Delphi’s Component palette and Object Inspector, you can place VCL components on forms and manipulate their properties without writing code.

All VCL objects descend from *TObject*, an abstract class whose methods encapsulate fundamental behavior like construction, destruction, and message handling. *TObject* is the immediate ancestor of many simple classes.

Components in the VCL descend from the abstract class *TComponent*. Components are objects that you can manipulate on forms at design time. Visual components—that is, components like *TForm* and *TSpeedButton* that appear on the screen at runtime—are called *controls*, and they descend from *TControl*.

Despite its name, the VCL consists mostly of nonvisual objects. The Delphi IDE allows you to add many nonvisual components to your programs by dropping them onto forms. For example, if you were writing an application that connects to a database, you might place a *TDataSource* component on a form. Although *TDataSource* is nonvisual, it is represented on the form by an icon (which doesn’t appear at runtime). You can manipulate the properties and events of *TDataSource* in the Object Inspector just as you would those of a visual control.

When you write classes of your own in Object Pascal, they should descend from *TObject*. By deriving new classes from the VCL’s base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with the VCL.

Using the object model

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you create an object (or, more formally, a class), you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

If you want to create new components and put them on the Delphi Component palette, see Chapter 31, “Overview of component creation.”

What is an object?

An object, or *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.

You can begin to understand objects if you understand Object Pascal *records*. Records (analogous to *structures* in C) 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 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, Delphi displays a new form for you to customize. In the Code editor, Delphi declares a new class type for the form and produces the code that creates the new form instance. The generated code looks like this:

```
unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Apps;

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 of implemntation 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 to the form any components (the fields of the new object) 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;

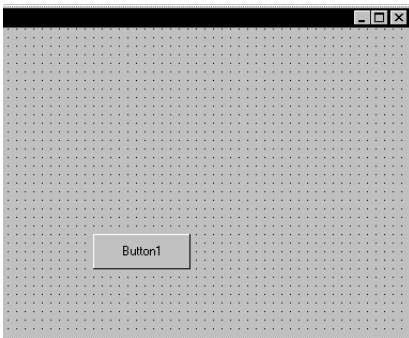
```

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 Delphi 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:

Figure 2.1 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, Classes, Graphics, Forms, Controls, Apps;

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 in Delphi 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 within 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 *ColorBox*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's.DFM file (which you usually don't edit manually) and in the Object Pascal source code that Delphi generates:

```

unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls, Apps;

type
  TColorBox = class(TForm){ Changed from TForm1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ColorBox: TColorBox;{ Changed from Form1 to ColorBox }

implementation

{$R *.DFM}

procedure TColorBox.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 TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;

```

Inheriting data and code from an object

The *TForm1* object described on page 2-2 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 VCL 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, Delphi automatically derives a new form from the *TForm* type:

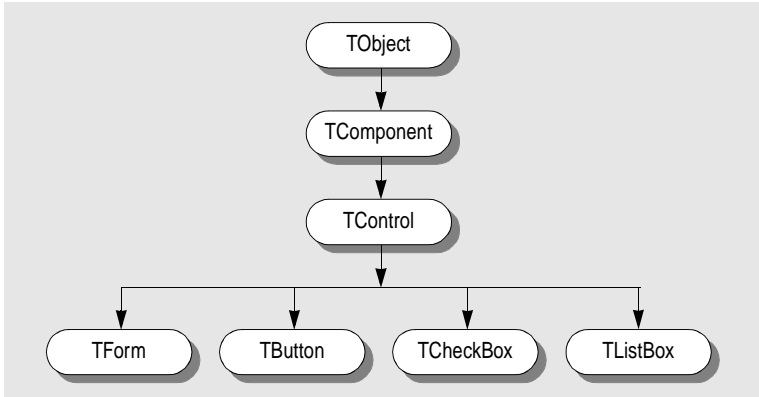
```
TForm1 = class(TForm)
```

A derived object inherits all the properties, events, and methods of the object it derives from. The derived object is called a *descendant* and the object it derives from 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. An object can have only one immediate ancestor, but it can have many direct descendants.

Objects, components, and controls

Figure 2.2 A simplified hierarchy diagram



The diagram above is a greatly simplified view of the inheritance hierarchy of the Visual Component Library. Every object inherits from *TObject*, and many objects inherit from *TComponent*. Controls, which inherit from *TControl*, have the ability to display themselves at runtime. A control like *TCheckBox* inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and adds specialized capabilities of its own.

Scope and qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared within an object are available to that object and its descendants. Although a method's implementation code appears outside of the object declaration, the method is still within the scope of the object because it is declared within the object's declaration.

When you write code to implement a method that refers to properties, methods, or fields of the object where the method is declared, you don't need to preface those identifiers with the name of the object. 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.

Delphi 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 an object extends to the object's descendants. You can, however, redeclare a field, property, or method within a descendant object. Such redeclarations either hide or override the inherited member.

For more information about scope, inheritance, and the **uses** clause, see the *Object Pascal Language Guide*.

Private, protected, public, and published declarations

When you declare a field, property, or method, the new member has a *visibility* indicated by one of the keywords **private**, **protected**, **public**, or **published**. The visibility of a member determines its accessibility to other objects and units.

- A private member is accessible only within the unit where it is declared. Private members are often used within a class to implement other (public or published) methods and properties.
- A protected member is accessible within the unit where its class is declared and within any descendant class, regardless of the descendant class's unit.
- A public member is accessible from wherever the object it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.
- 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.

For more information about visibility, see the *Object Pascal Language Guide*.

Using object variables

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable you are assigning to is an ancestor of the type of the variable being assigned. For example, here is a *TDataForm* type

declaration and a variable declaration section declaring two variables, *AForm* and *DataForm*:

```

type
  TDataForm = class(TForm)
  Button1: TButton;
  Edit1: TEdit;
  DataGrid1: TDataGrid;
  Database1: TDatabase;
private
  { Private declarations }
public
  { Public declarations }
end;

var
  AForm: TForm;
  DataForm: TDataForm;

```

AForm is of type *TForm*, and *DataForm* is of type *TDataForm*. Because *TDataForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := DataForm;
```

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 Delphi, 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 datasource components, have no visual representation at runtime.

You may want to create your own objects. For example, you could create a *TEmployee* object 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: Real;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Real read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Real;
  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 by Delphi. But 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* verifies that the object still exists before calling *Destroy*. For example,

```
Employee.Free
```

destroys the *Employee* object and deallocates its memory.

Components and ownership

Delphi has 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.

Note Don't confuse a component's *owner* with its *parent*. See "Parent properties" on page 2-11.

Using components

All components share features inherited from *TComponent*. By placing components on forms, you build the interface and functionality of your application. The standard components included with Delphi are sufficient for most application development, but you can extend the VCL by creating components of your own. For more information about creating custom components, see Chapter 31, “Overview of component creation.”

Delphi's standard components

The Component palette contains a selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette, and you can create component *templates* and *frames* that group several components.

The components on the palette are arranged in pages according to their purpose and functionality. Which pages appear in the default configuration depends on the version of Delphi you are running. Table 2.1 lists typical default pages and the kinds of components they contain.

Table 2.1 Component palette pages

Page name	Contents
Standard	Standard Windows controls, menus
Additional	Additional controls
Win32	Windows 9x/NT 4.0 common controls
System	Components and controls for system-level access, including timers, multimedia, and DDE
Internet	Components for internet communication protocols and Web applications
Data Access	Nonvisual components for accessing databases tables, queries, and reports
Data Controls	Visual, data-aware controls
Decision Cube	Controls that let you summarize information from databases and view it from a variety of perspectives
QReport	Quick Report components for creating embedded reports
Dialogs	Windows common dialog boxes
Win 3.1	Components for compatibility with Delphi 1.0 projects
Samples	Sample custom components
ActiveX	Sample ActiveX controls
Midas	Components used for creating multi-tiered database applications

The online Help provides information about the components on the default palette. The components on the ActiveX and Samples pages, however, are provided as examples only and are not documented.

Properties common to visual components

All visual components (descendants of *TControl*) share certain properties including

- Position and size properties
- Display properties
- Parent properties
- Navigation properties
- Drag-and-drop properties
- Drag-and-dock properties

While these properties are inherited from *TControl*, they are published—and hence appear in the Object Inspector—only for components to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays.

Position and size properties

Four properties define the position and size of a control on a form:

- *Height* sets the vertical size
- *Width* sets the horizontal size
- *Top* positions the top edge
- *Left* positions the left edge

These properties aren't accessible in nonvisual components, but Delphi does keep track of where you place the component icons on your forms. Most of the time you'll set and alter these properties by manipulating the control's image on the form or using the Alignment palette. You can, however, alter them at runtime.

Display properties

Four properties govern the general appearance of a control:

- *BorderStyle* specifies whether a control has a border.
- *Color* changes the background color of a control.
- *Ctrl3D* specifies whether a control will have a 3-D look or a flat border.
- *Font* changes the color, type family, style, or size of text.

Parent properties

To maintain a consistent appearance across your application, you can make any control look like its container—called its *parent*—by setting the *parent...* properties to *True*. For example, if you place a button on a form and set the button's *ParentFont* property to *True*, changes to the form's *Font* property will automatically propagate to the button (and to the form's other children). Later, if you change the button's *Font* property, your font choice will take effect and the *ParentFont* property will revert to *False*.

Note Don't confuse a component's *parent* with its *owner*. See "Components and ownership" on page 2-9.

Navigation properties

Several properties determine how users navigate among the controls in a form:

- *Caption* contains the text string that labels a component. To underline a character in a string, include an ampersand (&) before the character. This type of character is called an accelerator character. The user can then select the control or menu item by pressing *Alt* while typing the underlined character.
- *TabOrder* indicates the position of the control in its parent's tab order, the order in which controls receive focus when the user presses the *Tab* key. Initially, tab order is the order in which the components are added to the form, but you can change this by changing *TabOrder*. *TabOrder* is meaningful only if *TabStop* is *True*.
- *TabStop* determines whether the user can tab to a control. If *TabStop* is *True*, the control is in the tab order.

Drag-and-drop properties

Two component properties affect drag-and-drop behavior:

- *DragMode* determines how dragging starts. By default, *DragMode* is *dmManual*, and the application must call the *BeginDrag* method to start dragging. When *DragMode* is *dmAutomatic*, dragging starts as soon as the mouse button goes down.
- *DragCursor* determines the shape of the mouse pointer when it is over a draggable component.

Drag-and-dock properties

The following properties control drag-and-dock behavior.

- *DockSite*
- *DragKind*
- *DragMode*
- *FloatingDockSiteClass*
- *AutoSize*

For more information, see “Implementing drag-and-dock in controls” on page 6-4.

Text controls

Many applications present text to the user or 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:
Edit	Edit a single line of text
Memo	Edit multiple lines of text
MaskEdit	Adhere to a particular format, such as a postal code or phone number
RichEdit	Edit multiple lines of text using rich text format

Properties common to all text controls

All of the text controls have these properties in common:

- *Text* determines the text that appears in the edit box or memo control.
- *CharCase* forces the case of the text being entered to lowercase or uppercase.
- *ReadOnly* specifies whether the user is allowed to change the text.
- *MaxLength* limits the number of characters in the control.

- *PasswordChar* hides the text by displaying a single character (usually an asterisk).
- *HideSelection* specifies whether selected text remains highlighted when the control does not have focus.

Properties shared by memo and rich text controls

Memo and rich text controls, which handle multiple lines of text, have several properties in common:

- *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.
- *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.
- *SelText* contains the currently selected (highlighted) part of the text.
- *SelStart* and *SelLength* indicate the position and length of the selected part of the text.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Rich text controls

The rich edit component 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.

Specialized input controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
ScrollBar	Select values on a continuous range
TrackBar	Select values on a continuous range (more visually effective than scroll bar)
UpDown	Select a value from a spinner attached to an edit component
HotKey	Enter <i>Ctrl/Shift/Alt</i> keyboard sequences

Scroll bars

The scroll bar component is a Windows 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, since many visual components provide scroll bars of their own that don't require additional coding. For example,

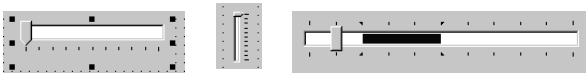
TForm has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBar*.

Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like 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. See Figure 2.3. 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 *SetTicks* method.

Figure 2.3 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

An up-down control 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

Use the hot key component 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*.

Splitter control

A splitter 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 invoke a command in an application. Delphi offers several button-like controls:

Use this component:	To do this:
Button	Present command choices on buttons with text
BitBtn	Present command choices on buttons with text and glyphs
SpeedButton	Create grouped toolbar buttons
CheckBox	Present on/off options
RadioButton	Present a set of mutually exclusive choices
ToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions
CoolBar	Display a collection of windowed controls within movable, resizable bands

Button controls

Users click button controls to initiate actions. 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 (*BitBtn*) 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 is to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered in 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, 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 presents the user with two, or sometimes three, choices.

- 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*).

Radio buttons

Radio buttons present a set of mutually exclusive choices. You can use individual radio buttons or the *radio group* component, which arranges groups of radio buttons automatically. See “Grouping components” on page 2-18 for more information.

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 *ToolBar* component, then right-click and choose New Button to add buttons to the toolbar. The *ToolBar* 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 the *ToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

Cool bars

A cool bar (or rebar) 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.

- 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 simply by adding new windowed controls from the palette.
- The *FixedOrder* property determines users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

Handling lists

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
ListBox	A list of text strings
CheckBoxListBox	A list with a check box in front of each item
ComboBox	An edit box with a scrollable drop-down list
TreeView	A hierarchical list
ListView	A list of (draggable) items with optional icons, columns, and headings
DateTimePicker	A list box for entering dates or times
MonthCalendar	A calendar for selecting dates

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” on page 2-28.

List boxes and check-list boxes

List boxes and check-list boxes display lists from which users can select items.

- *Items* uses a *TStringList* 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.
- *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” on page 6-11.

Combo boxes

A combo box 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.

Use the *Style* property to select the type of combo box you need:

- Use *csDropDown* if you want an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.

- 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.
- 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” on page 6-11.

Tree views

A tree view 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 a 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.
- *ShowRoot* determines whether lines connecting the top-level items are displayed.

List views

List views 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.
- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The left most column contains a small icon and label, and subsequent columns contain sub items specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

Date-time pickers and month calendars

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 a recent version of COMCTL32.DLL (usually located in the WINDOWS\SYSTEM or WINDOWS\SYSTEM32 directory) at both design time and runtime.

Grouping components

A graphical interface is easier to use when related controls and information are presented in groups. Delphi provides several components for grouping components:

Use this component:	When you want this:
GroupBox	A standard group box with a title
RadioGroup	A simple group of radio buttons
Panel	A more visually flexible group of controls

Use this component:	When you want this:
ScrollBox	A scrollable region containing controls
TabControl	A set of mutually exclusive notebook-style tabs
PageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
HeaderControl	Resizable column headers

Group boxes and radio groups

A group box is a standard Windows component that 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 Component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component 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 panel component provides a generic container for other controls. 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.

Scroll boxes

Scroll boxes 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. 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. 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.

Tab controls

The tab control component looks 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 is a page set suitable for multipage dialog boxes. To create a new page in a page control, right-click the control and choose New Page.

Header controls

A header control 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.

Visual feedback

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.

Use this component or property:	To do this:
Label and <i>StaticText</i>	Display non-editable text
<i>StatusBar</i>	Display a status region (usually at the bottom of a window)
<i>ProgressBar</i>	Show the amount of work completed for a particular task
<i>Hint</i> and <i>ShowHint</i>	Activate fly-by or "tool-tip" help
<i>HelpContext</i> and <i>HelpFile</i>	Link context-sensitive online Help

Labels and static-text components

Labels display text and are usually placed next to other controls. The standard label component, *TLabel*, is a non windowed control, 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.
- *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.

Status bars

Although you can use a panel to make a status bar, it is simpler to use the status-bar component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

You will usually divide a status bar into several text areas. To create text areas, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. The *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 to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.

Figure 2.4 A progress bar



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 *TDBGrid* or *TDBCtrlGrid* component described in Chapter 26, "Using data controls". 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.

Graphic display

The following components make it easy to incorporate graphics into an application.

Use this component	To display:
Image	Graphics files
Shape	Geometric shapes
Bevel	3D lines and frames
PaintBox	Graphics drawn by your program at runtime
Animate	AVI files

Images

The image component 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.

Shapes

The shape component displays a geometric shape. It is a non windowed control and 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 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 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" on page 7-1.

Animation control

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip. 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 or compressed using run-length encoding (RLE).

Windows common dialog boxes

The components on the Dialogs page of the Component palette make the Windows common dialog boxes available in Delphi applications. These dialog boxes provide a consistent user interface for standard operations like finding and opening files, setting fonts and colors, and printing. The dialogs do not appear at runtime until activated by a call to their *Execute* method.

Setting component properties

Published properties can be set at design time in the Object Inspector and, in some cases, with special property editors.

To set properties at runtime, assign them new values in your application source code. For information about the properties of each component, see the VCL Help.

Using the Object Inspector

When you select a component on a form, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the Value column and the Property 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 displays a list of sub values for the property.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see “property categories” in the online Help.

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.

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 or click the ellipsis mark. 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:

```
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:

```
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:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Repaint;
end;
```

For more information about scope, see “Scope and qualifiers” on page 2-6.

Working with events and event handlers

In Delphi, 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 an Object Pascal 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

Delphi 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*. Delphi generates the event handler in the Code editor and places the cursor inside the **begin...end** block.

- 4 Inside the **begin...end** block, 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 the Bevel, don't respond to any events. Other components respond differently when you double-click on 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 at the beginning of the event-handler body.

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 procedure above is an easy way to reuse event handlers. *Action lists*, however, provide a more powerful tool for centrally organizing the code that responds to user

commands. For more information about action lists, see “Using actions” on page 5-36.

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

Delphi’s 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 section explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see “Creating and managing menus” on page 5-15.

To create an event handler for a menu item,

- 1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
- 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. Delphi generates an event handler in the Code editor and places the cursor inside the **begin...end** block.
- 4 Inside the **begin...end** block, 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 on a *MainMenu* or *PopupMenu* object.

- 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 using the Form Designer, Delphi 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 **interface** section of the unit) and its implementation (in the **implementation** section); otherwise you'll get a compiler error when you build your project.

Using helper objects

The VCL includes a variety of nonvisual objects that simplify common programming tasks. This section describes a few Helper objects that facilitate

- Creating and managing lists
- Creating and managing string lists
- Editing the Windows registry and .INI files
- Streaming data to a hard disk or other storage device

Working with lists

Several VCL objects provide functionality for creating and managing lists:

- *TList* maintains a list of pointers.
- *TObjectList* maintains a memory-managed list of instance objects.
- *TComponentList* maintains a memory-managed list of components (that is, instances of classes descended from *TComponent*).
- *TQueue* maintains a first-in first-out list of pointers.
- *TStack* maintains a last-in first-out list of pointers.
- *TObjectQueue* maintains a first-in first-out list of objects.
- *TObjectStack* maintains a last-in first-out list of objects.
- *TClassList* maintains a list of class types.
- *TCollection*, *TOwnedCollection*, and *TCollectionItem* maintain indexed collections of specially defined items.
- *TStringList* maintains a list of strings.

For more information about these objects, see the VCL Reference in the online Help.

Working with string lists

Applications often need to manage lists 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 provides a common interface to any list of strings through an object called *TStrings* and its descendant *TStringList*. In addition to providing functionality for maintaining string lists, these objects allow easy inter operability; for example, you can edit the lines of a memo (which are an instance of *TStrings*) and then use these lines as items in a combo box (also an instance of *TStrings*).

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 WIN.INI file into a memo field and makes a backup copy called WIN.BAK.

```

procedure EditWinIni;
var
  FileName: string;{ storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.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;

```

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.

- 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.

```

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;

```

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.

- 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. (*OnCreate* is the default event for a form, so just double-click on the form to generate a skeleton event handler.) The *OnCreate* event handler, which executes before the form appears, 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 *OnDestroy* 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.

```

unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

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: TStringList;{ 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.

```

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 array property *Strings* contains the strings in the list, referenced by a zero-based index. Since *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus


```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

Finding the position of a string in the 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:

```
if FileListBox1.Items.IndexOf('WIN.INI') > -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* $- 1$.

This example converts each string in a list box to uppercase characters.

```
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;
```

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

```
Insert(2, 'Three');
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2); { append the strings from StringList2 to StringList1 }
```

Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use

```
Move(2, 4)
```

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.

This example uses *IndexOf* and *Delete* find and delete a string.

```
with ListBox1.Items do
begin
  if IndexOf('bureaucracy') > -1 then
    Delete(IndexOf('bureaucracy'));
end;
```

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,

```
Memo1.Lines.Assign(ComboBox1.Items); { overwrites original strings }
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items); { appends strings to end }
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you simply assign one string-list variable to another—

```
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.

The Windows registry and INI files

The Windows system registry is a hierarchical database where applications store configuration information. The VCL object *TRegistry* supplies methods that read and write to the registry.

Until Windows 95, most applications stored configuration information in initialization files, usually named with the extension .INI. The VCL provides objects that facilitate maintenance and migration of programs that use INI files. Use

- *TRegistry* to work with the registry.

- *TIniFile* or *TMemIniFile* to work with Windows 3.x INI files.
- *TRegistryIniFile* when you want to work with both the registry and INI files. *TRegistryIniFile* has properties and methods similar to those of *TIniFile*, but it reads and writes to the system registry. By using a variable of type *TCustomIniFile* (the common ancestor of *TIniFile*, *TMemIniFile*, and *TRegistryIniFile*), you can write generic code that accesses either the registry or an INI file, depending on where it is called.

Using streams

Use specialized stream objects to read or write to storage media. Each descendant of *TStream* implements methods for accessing a particular medium, such as disk files, dynamic memory, and so on. *TStream* descendants include *TFileStream*, *TStringStream*, *TMemoryStream*, *TBlobStream*, and *TWinSocketStream*. In addition to methods for reading and writing, these objects permit applications to seek to an arbitrary position in the stream. Properties of *TStream* provide information about the stream, such as size and current position.

Using data modules and remote data modules

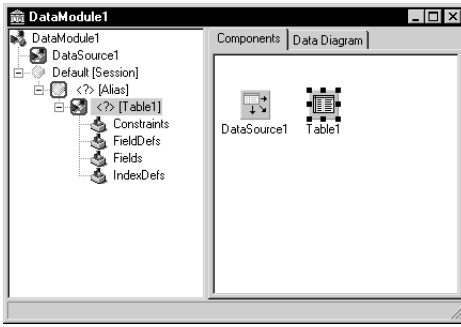
A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are two types of data module: standard and remote. To create a single- or two-tiered application, use a standard data module. If you have the Client/Server or Enterprise edition of Delphi and are creating a multi-tiered application, you can add a remote data module to your application server; see “Adding a remote data module to an application server project” on page 2-34.

Creating and editing data modules

To create a data module, choose File | New and double-click on Data Module. Delphi opens an empty data module in the Data Module Designer, displays the unit file for the new module in the Code editor, and adds the module to the current project. When you reopen an existing data module, Delphi displays its components in the Data Module Designer.

The Data Module Designer is divided into two panes. The left pane displays a hierarchical tree view of the components in the module. The right pane has two tabs: Components and Data Diagram. The Components page shows the components as they would appear on a form. The Data Diagram page shows a graphical representation of internal relationships among the components, such as master-detail links and lookup fields.

Figure 2.5 A simple data module

You can add components to a data module by selecting them on the Component palette and clicking in the Tree or Components view of the Data Module Designer. When a component is selected in the Data Module Designer, you can edit its properties in the Object Inspector just as you would if the component were on a form. For more information about the Data Module Designer, see the online Help.

Creating business rules in a data module

In a data module's unit file, you can write methods, including event handlers for the components in the module, as well as global routines that encapsulate business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping; you could call such a procedure from an event handler for a component in the module or from any unit that uses 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.
- Choose File | Use Unit, then enter the name of the module or pick it from the list box in the Use Unit dialog.
- Double-click on a *TTable* or *TQuery* component in the data module to open the Fields editor. From the Fields editor, drag any fields onto your form. Delphi 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.

Adding a remote data module to an application server project

Some versions 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, choose File | New, select the Multitier page in the New Items dialog box, and double-

click the desired type of module (Remote Data Module, MTS Data Module, or CORBA Data Module) to open the Remote Data Module wizard. Once you add a remote data module to a project, you use it just like a standard data module.

For more information about multi-tiered database applications, see Chapter 14, “Creating multi-tiered applications.”

Using the Object Repository

The Object Repository (Tools | Repository) 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 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), 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, Delphi 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 | 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, Delphi creates a DELPHI32.DRO file 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. 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 in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository 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 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.

Delphi copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

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.

Specifying a default project, new form, and main form

By default, when you choose File | New Application or File | New Form, Delphi displays a blank form. You can change this behavior by reconfiguring the Repository:

- 1 Choose Tools | Repository
- 2 If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.
- 3 If you want to specify a default form, select a Repository page (such as Forms), then choose a form under Objects. To specify the default new form (File | New Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.
- 4 Click OK.

Adding custom components to the IDE

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a component, see Part IV, “Creating custom components”. To install an existing component, see “Installing component packages” on page 9-6.

Common programming tasks

This chapter discusses the fundamentals for some of the common programming tasks in Delphi:

- Handling exceptions
- Using interfaces
- Working with strings
- Working with files

Handling exceptions

Delphi provides a mechanism to ensure that applications are robust, meaning that they handle errors in a consistent manner. Exception handling allows the application to recover from errors if possible and to shut down if need be, without losing data or resources. Error conditions in Delphi are indicated by exceptions. This section describes the following tasks for using exceptions to create safe applications:

- Protecting blocks of code
- Protecting resource allocations
- Handling RTL exceptions
- Handling component exceptions
- Using `TApplication.HandleException`
- Silent exceptions
- Defining your own exceptions

Protecting blocks of code

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 will present 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.

To protect blocks of code you need to understand

- Responding to exceptions
- Exceptions and the flow of control
- Nesting exception responses

Responding to exceptions

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

- **Executing cleanup code:** The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state. You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.
- **Handling an exception:** This is 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, might be more difficult for the application or the user to correct.

Exceptions and the flow of control

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

Example The following code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```

...
try{ begin the protected block }
    Font.Name := 'Courier';{ if any exception occurs... }
    Font.Size := 24;{ ...in any of these statements... }
    Color := clBlue;
except{ ...execution jumps to here }
    on Exception do MessageBeep(0);{ this handles any exception by beeping }
end;
...{ execution resumes here, outside the protected block}

```

Nesting exception responses

Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks inside other blocks, you can customize responses even within blocks that already customize responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:

```

                { allocate first resource }
                try
                { allocate second resource }
                try
                { code that uses both resources }
                finally
                { release second resource }
                end;
                finally
                { release first resource }
                end;

```

Diagram illustrating nested blocks:

- The outermost block is labeled "protected block" on the left.
- Inside it is a "try" block.
- Inside the "try" block is a "nested protected block" (labeled on the left).
- Inside the "nested protected block" is another "try" block.
- Inside the inner "try" block is "code that uses both resources".
- Following the inner "try" block is a "finally" block containing "{ release second resource }".
- Following the "nested protected block" is a "finally" block containing "{ release first resource }".
- The entire structure ends with an "end;" statement.

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:

```

                try
                { protected code }
                try
                { specially protected code }
                except
                { local exception handling }
                end;
                except
                { global exception handling }
                end;

```

Diagram illustrating nested exception handling:

- The outermost block is labeled "exception-handling block" on the left.
- Inside it is a "try" block.
- Inside the "try" block is "protected code".
- Inside the "try" block is a "nested exception-handling block" (labeled on the left).
- Inside the "nested exception-handling block" is another "try" block.
- Inside the inner "try" block is "specially protected code".
- Following the inner "try" block is an "except" block containing "{ local exception handling }".
- Following the "nested exception-handling block" is an "except" block containing "{ global exception handling }".
- The entire structure ends with an "end;" statement.

You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

Protecting resource allocations

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- What kind of resources need protection?
- Creating a resource protection block

What kind of resources need protection?

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
- Objects

Example The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```

procedure TForm1.Button1Click(Sender: TComponent);
var
    APointer: Pointer;
    AnInteger, ADividend: Integer;
begin
    ADividend := 0;
    GetMem(APointer, 1024);{ allocate 1K of memory }
    AnInteger := 10 div ADividend;{ this generates an error }
    FreeMem(APointer, 1024);{ it never gets here }
end;

```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the *FreeMem* statement never gets to free the memory.

In order to guarantee that the *FreeMem* gets to free the memory allocated by *GetMem*, you need to put the code in a resource-protection block.

Creating a resource protection block

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```
{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;
```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) 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 is executed, 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** part.

Example The following code illustrates an event handler that allocates memory and generates an error, but still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the error }
  end;
end;
```

The statements in the termination code do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the termination code.

Handling RTL exceptions

When you write code that calls routines in the runtime library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also silent exceptions that do not, by default, display a message.

To handle RTL exceptions effectively, you need to understand the following:

- What are the RTL exceptions?
- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

What are the RTL exceptions?

The runtime library's exceptions are defined in the *SysUtils* unit, and they all descend from a generic exception-object type called `Exception`. `Exception` provides the string for the message that RTL exceptions display by default.

There are several kinds of exceptions raised by the RTL, as described in the following table.

Table 3.1 RTL exceptions

Error type	Cause	Meaning
Input/output	Error accessing a file or I/O device	Most I/O exceptions are related to error codes returned by Windows when accessing a file.
Heap	Error using dynamic memory	Heap errors can occur when there is insufficient memory available, or when an application disposes of a pointer that points outside the heap.
Integer math	Illegal operation on integer-type expressions	Errors include division by zero, numbers or expressions out of range, and overflows.
Floating-point math	Illegal operation on real-type expressions	Floating-point errors can come from either a hardware coprocessor or the software emulator. Errors include invalid instructions, division by zero, and overflow or underflow.
Typecast	Invalid typecasting with the <code>as</code> operator	Objects can only be typecast to compatible types.
Conversion	Invalid type conversion	Type-conversion functions such as <code>IntToStr</code> , <code>StrToInt</code> , and <code>StrToFloat</code> raise conversion exceptions when the parameter cannot be converted to the desired type.
Hardware	System condition	Hardware exceptions indicate that either the processor or the user generated some kind of error condition or interruption, such as an access violation, stack overflow, or keyboard interrupt.
Variant	Illegal type coercion	Errors can occur when referring to variants in expressions where the variant cannot be coerced into a compatible type.

For a list of the RTL exception types, see the *SysUtils* unit.

Creating an exception handler

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, 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.

Exception handling statements

Each **on** statement in the **except** part of a **try..except** block defines code for handling a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

Example You can define an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems; { handle the normal case }
  except
    on EDivByZero do Result := 0; { handle the exception only if needed }
  end;
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then { always test }
    Result := Sum div NumberOfItems { use normal calculation }
  else Result := 0; { handle exceptional case }
end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation 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 single time to make sure you're allowed to proceed with each step in the calculation.

Using the exception instance

Most of the time, an 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.

Example If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

The temporary variable (E in this example) is of the type specified after the colon (*EInvalidOperation* in this example). You can use the `as` operator to typecast the exception into a more specific type if needed.

Note 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.

Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. In fact, you need to provide handlers only for those 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 the block (or 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.

Providing default exception handlers

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an `else` part to the **except** part of 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 the containing block.

Caution 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 use an enclosing **try..finally** block:

```
try
  try
    { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;
```

For another approach to augmenting exception handling, see Reraising the exception.

Handling classes of exceptions

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

Example The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

Reraising the exception

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing 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.

Example When an exception occurs, you might want to display some sort of message to the user, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word `raise`. This is called reraising the exception, as shown in this example:

```
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;
```

If code in the `{ statements }` part raises an *ESomething* exception, only the handler in the outer **except** part executes. However, if code in the `{ special statements }` part raises *ESomething*, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

Handling component exceptions

Delphi's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a runtime error. The mechanics of handling component exceptions are no different than handling RTL exceptions.

Example 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 an "Index out of range" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```

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 in list box }
except
    on EStringListError do
        MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
end;
end;

```

If you click the button once, the list box has only three strings, so accessing the fourth string (Items[3]) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

Using TApplication.HandleException

HandleException provides default handling of exceptions for the application. If an exception passes through all the **try** blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating that an error has occurred. You can use *HandleException* in this fashion:

```

try
    { statements }
except
    Application.HandleException(Self);
end;

```

For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. Therefore, if you want to both handle the exception, and provide this default behavior as the VCL does, you can add a call to *HandleException* to your code:

```

try
    { special statements }
except
    on ESomething do
        begin
            { handling for only the special statements }
        end

```

```

        Application.HandleException(Self);{ call HandleException }
    end;
end;

```

For more information, search for exception handling routines in the Help index.

Silent exceptions

Delphi 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 handle an exception, 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 Delphi 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 an *EAbort* exception.

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 will break out of the current operation without displaying an error message.

Example 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:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
begin
    for I := 1 to 10 do{ loop ten times }
        begin
            ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
            if I = 7 then Abort;{ abort after the seventh one }
        end;
    end;
end;

```

Defining your own exceptions

In addition to protecting your code from exceptions generated by the runtime library and various components, you can use the same mechanism to manage exceptional conditions in your own code.

To use exceptions in your code, you need to understand these steps:

- Declaring an exception object type
- Raising an exception

Declaring an exception object type

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from `Exception`.

It's therefore a good idea to derive any new exception types 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 a specific exception handler for that exception, one of the standard handlers will handle it instead.

Example For example, consider the following declaration:

```
type
    EMyException = class(Exception);
```

If you raise `EMyException` but don't provide a specific handler for `EMyException`, 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 could at least see that it was your new exception raised.

Raising an exception

To indicate an error condition in an application, you can raise an exception which involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word `raise`, followed by an instance of an exception object. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Setting the exception address is done through the `ErrorAddr` variable in the `System` unit. Raising an exception sets this variable to the address where the application raised the exception. You can refer to `ErrorAddr` in your exception handlers, for example, to notify the user of where the error occurred. You can also specify a value for `ErrorAddr` when you raise an exception.

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.

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');
```

Using interfaces

Delphi's **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of the VCL by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Interfaces are useful when sets of operations, such as streaming, are used across a broad range of objects. Interfaces are also a fundamental aspect of the COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture) distributed object models.

Interfaces as a language feature

An interface is like a class that contains only abstract methods and a clear definition of their functionality. Strictly speaking, interface method definitions include the number and types of their parameters, their return type, and their expected behavior. Interface methods are semantically or logically related to indicate the purpose of the interface. It is convention for interfaces to be named according to their behavior and to be 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. A simple example of declaring an interface is:

```
type
  IEdit = interface
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
  end;
```

Like abstract classes, interfaces themselves can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, you must 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; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
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.

Sharing interfaces between classes

Using interfaces offers a design approach to separating the way a class is used from the way it is implemented. Two classes can share the same interface without requiring that they descend from the same base class. This polymorphic invocation of the same methods on unrelated objects is possible as long as 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 say, *TFigure* which implemented a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. The above *IPaint* would be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

which makes sense for the rectangle to support 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 used to fill 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 *IUnknown* that manage reference counting. For more information, see “Implementing IUnknown” on page 3-16 and “Memory management of interface objects” on page 3-20.

Using interfaces with procedures

Interfaces also allow you to write generic procedures that can handle objects without requiring the objects to descend from a particular base class. Using the above *IPaint* and *IRotate* interfaces 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 above objects to be used more often than if they were written to only work against a *TFigure* class.

Form details about the syntax, language definitions and rules for interfaces, see the *Object Pascal Language Guide* online Help section on Object interfaces.

Implementing IUnknown

All interfaces derive either directly or indirectly from the *IUnknown* interface. This interface provides the essential functionality of an interface, that is, dynamic querying and lifetime management. This functionality is established in the three *IUnknown* methods:

- *QueryInterface* provides a method for dynamically querying a given object and obtaining interface references for the interfaces the object supports.
- *AddRef* is a reference counting method that increments the count each time the call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *Release* is used in conjunction with *AddRef* to enable an object to track its own lifetime and to determine when it is safe to delete itself. Once the reference count reaches zero the interface implementation releases the underlying object(s).

Every class that implements interfaces must implement the three *IUnknown* 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.

TInterfacedObject

The VCL defines a simple class, *TInterfacedObject*, that serves as a convenient base because it implements the methods of *IUnknown*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendent of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
  ...
  end;
```

Because it implements the methods of *IUnknown*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see “Memory management of interface objects” on page 3-20, 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

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 *IUnknown* interface since 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 *IUnknown*.

When you use the **as** operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IUnknown*: Although all interfaces derive from *IUnknown*, it is not sufficient, if you want to use the **as** operator, for a class to simply implement the methods of *IUnknown*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IUnknown* in its ancestor 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 an object contain, or be contained by another. The VCL uses properties that are object types as an approach to containment and code reuse. To support this design for interfaces Delphi has a keyword **implements**, that makes it easy to write code to delegate all or part of the implementation of an interface to a sub-object. Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object contains an inner object that implements interfaces which are exposed only by the outer object. The VCL has classes that support aggregation.

Using implements for delegation

Many classes in the VCL have properties that are sub-objects. 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 property instance. 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 keyword **implements** delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods as a way of overriding 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;

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.

```

For more information about the syntax, implementation details, and language rules of the **implements** keyword, see the *Object Pascal Language Guide* online Help section on Object interfaces.

Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that define 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. The only requirement is that it implement *IUnknown*. The inner object, or objects, can implement one or more interfaces, however only the outer object exposes the interfaces. These include both the interfaces it implements and the ones implemented by its contained objects. 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 implementation model for aggregation defines explicit rules for implementing *IUnknown* using delegation. The inner object must implement an *IUnknown* on itself, that controls the inner object's reference count. This implementation of *IUnknown* 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 *IUnknown*. The inner object also implements a second *IUnknown* for all the interfaces it implements. These are the interfaces exposed by the outer object. This second *IUnknown* delegates calls to *QueryInterface*, *AddRef*, and *Release* to the outer object. The outer *IUnknown* 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 *IUnknown* 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 *IUnknown* provide a way of implementing this functionality. Their defined behavior states that they will 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, 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

Delphi provides most of the *IUnknown* 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 these classes. *TInterfacedObject* is the non-CoClass that provides this behavior. 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, will increment the reference count (call *AddRef*) on the parameter, whereas either of the following declarations:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

will not. 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 VCL component or a control that is owned by another component, then your object is part of a different memory management system that is based in *TComponent*. You should not mix the object lifetime management approaches of VCL components and COM reference counting. If you want to create a component that supports interfaces, you can implement the *IUnknown AddRef* and *Release* methods as empty functions to bypass the COM reference counting mechanism:

```
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 do implement *QueryInterface*, you can still use the **as** operator for interfaces on components, 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, component object that the decision is made to circumvent the *AddRef* and *Release* methods, and to handle memory management via the VCL approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a component as its containing outer object.

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. For more information distinguishing the various implementations of the *IUnknown* interface by the inner and outer objects, see “Aggregation” on page 3-19 and the Microsoft online Help topics on the “controlling Unknown.”

Using interfaces in distributed applications

Interfaces are a fundamental element in the COM and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IUnknown* interface methods.

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 Chapter 44, “Overview of COM technologies.”

The other distributed application technology is CORBA. The use of interfaces in CORBA applications is mediated by stub classes on the client and skeleton classes on the server. These stub and skeleton classes handle the details of marshaling interface calls so that parameter values and return values can be transmitted correctly. Applications must use either a stub or skeleton class, or employ the Dynamic Invocation Interface (DII) which converts all parameters to special variants (so that they carry their own type information.)

Although it is not a necessary feature of CORBA technology, Delphi implements CORBA using class factories, similar to the way in which COM uses class factories and CoClasses. By unifying the two distributed model architectures in this way, Delphi supports a combined COM/CORBA server that can service both COM and CORBA clients simultaneously. For more information about using interfaces with CORBA, see Chapter 28, “Writing CORBA applications.”

Working with strings

Delphi has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section is an overview of these types, their purpose, and usage. For language details, see the Object Pascal Language online Help on String types.

Character types

Delphi has three character types: *Char*, *AnsiChar*, and *WideChar*.

The *Char* character type came from Standard Pascal, and was used in Turbo Pascal and then in Object Pascal. Later Object Pascal added *AnsiChar* and *WideChar* as specific character types that were used to support standards for character representation on the Windows operating system. *AnsiChar* was introduced to support an 8-bit character ANSI standard, and *WideChar* was introduced to support a 16-bit Unicode standard. The name *WideChar* is used because Unicode characters are also known as wide characters. Wide characters are two bytes instead of one, so that the character set can represent many more different characters. When *AnsiChar* and *WideChar* were implemented, *Char* became the default character type representing the currently recommended implementation. If you use *Char* in your application, remember that its implementation is subject to change in future versions of Delphi.

The following table summarizes these character types:

Table 3.2 Object Pascal character types

Type	Bytes	Contents	Purpose
Char	1	Holds a single ANSI character.	Default character type.
AnsiChar	1	Holds a single ANSI character.	8-bit Ansi character standard on Windows.
WideChar	2	Holds a single Unicode character.	16-bit Unicode standard on Windows.

For more information about using these character types, see the *Object Pascal Language Guide* online Help on Character types. For more information about Unicode characters, see the *Object Pascal Language Guide* online Help on About extended character sets.

String types

Delphi has three categories of types that you can use when working with strings. These are character pointers, string types, and VCL string classes. This section summarizes string types, and discusses using them with character pointers. For information about using VCL string classes, see the online Help on TStrings.

There are currently three string implementations in Delphi: short strings, long strings, and wide strings. There are several different string types that represent these implementations. In addition, there is a reserved word **string** that defaults to the currently recommended string implementation.

Short strings

String was the first string type used in Turbo Pascal. **String** was originally implemented as a short string. Short strings are an allocation of between 1 and 256 bytes, of which the first byte contains the length of the string and the remaining bytes contain the characters in the string:

```
S: string[0..n]// the original string type
```

When long strings were implemented, **string** was changed to a long string implementation by default and *ShortString* was introduced as a backward compatibility type. *ShortString* is a predefined type for a maximum length string:

```
S: string[255]// the ShortString type
```

The size of the memory allocated for a *ShortString* is static, meaning that it is determined at compile time. However, the location of the memory for the *ShortString* can be dynamically allocated, for example if you use a *PShortString*, which is a pointer to a *ShortString*. The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one. For the *ShortString* predefined type the size is 256 bytes.

Both short strings, declared using the syntax **string[0..n]**, and the *ShortString* predefined type exist primarily for backward compatibility with earlier versions of Delphi and Borland Pascal.

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. The {\$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. However, short strings can be useful in data structures where you need a fixed-size component or in DLLs when you don't want to use the *ShareMem* unit (see also the online Help on Memory Management). You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to **string[255]** or *ShortString*, which are unambiguous and independent of the \$H setting.

For details about short strings and the *ShortString* type, see the *Object Pascal Language Guide* online Help on Short strings.

Long strings

Long strings are dynamically-allocated strings with a maximum length limited only by available memory. Like short strings, long strings use 8-bit Ansi characters and have a length indicator. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure. Long strings are also reference-counted and, like *PChars*, long strings are null-terminated. For details about the implementation of long strings, see the *Object Pascal Language Guide* online Help on Long strings.

Long strings are denoted by the reserved word **string** and by the predefined identifier *AnsiString*. For new applications, it is recommended that you use the long string type. All components in the Visual Component Library are compiled in this state, typically using **string**. If you write components, they should also use long

strings, as should any code that receives data from VCL string-type properties. If you want to write specific code that always uses a long string, then you should use *AnsiString*. If you want to write flexible code that allows you to easily change the type as new string implementations become standard, then you should use **string**.

WideString

The *WideChar* type allows wide character strings to be represented as arrays of *WideChars*. Wide strings are strings composed of 16-bit Unicode characters. As with long strings, *WideStrings* are dynamically allocated with a maximum length limited only by available memory. However, wide strings are not reference counted. The dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope. In all other respects wide strings possess the same attributes as long strings. The *WideString* type is denoted by the predefined identifier *WideString*.

Since the 32-bit version of OLE uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

For more information about *WideStrings*, see the *Object Pascal Language Guide* online Help on *WideString*.

PChar types

A *PChar* is a pointer to a null-terminated string of characters of the type *Char*. Each of the three character types also has a built-in pointer type:

- A *PChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PAnsiChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PWideChar* is a pointer to a null-terminated string of 16-bit characters.

PChars are, with short strings, one of the original Object Pascal string types. They were created primarily as a C language and Windows API compatibility type.

OpenString

An *OpenString* is obsolete, but you may see it in older code. It is for 16-bit compatibility and is allowed only in parameters. *OpenString* was used, before long strings were implemented, to allow a short string of an unspecified length string to be passed as a parameter. For example, this declaration:

```
procedure a(v : openstring);
```

will allow any length string to be passed as a parameter, where normally the string length of the formal and actual parameters must match exactly. You should not have to use *OpenString* in any new applications you write.

Refer also to the {\$P+/-} switch in “Compiler directives for strings” on page 3-31.

Runtime library string handling routines

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 *PChar* types use the null-termination to determine the length of the string. For more details about null-terminated strings, see Working with null-terminated strings in the *Object Pascal Language Guide* online Help.

The runtime library also includes a category of string formatting routines. 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.

Because wide strings and long strings are the commonly used types, the remaining sections discuss these routines.

Wide character routines

When working with strings you should make sure that the code in your application can handle the strings it will encounter in the various target locales. Sometimes you will need to use wide characters and wide strings. In fact, one approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. The runtime library includes the following wide character string functions for converting between standard single-byte character strings (or MBCS strings) and Unicode strings:

- `StringToWideChar`
- `WideCharLenToString`
- `WideCharLenToStrVar`
- `WideCharToString`
- `WideCharToStrVar`

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.

A disadvantage of working with wide characters is that Windows 95 does not support 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 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*.

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 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

Where appropriate, the tables also provide columns indicating whether or not a routine satisfies the following criteria.

- Uses case sensitivity: If the Windows locale is used, it determines the definition of case. If the routine does not use the Windows locale, analysis 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 the Windows locale: Windows locale enablement allows you to add extra features to your application for specific locales. In particular, for Asian language environments. Most Windows locales 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 Windows 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 as a mix of one and two byte character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS are written parse one- and two-byte characters. The *ByteType* and *StrByteType* determine whether a particular byte is the lead byte of a two-byte character. Be careful when using multi-byte characters not to truncate a string by cutting a two-byte 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 character or string. For more information about MBCS, see “Enabling application code” on page 10-2 of Chapter 10, “Creating international applications.”

TABLE comparison

Table 3.3 String comparison routines

Routine	Case-sensitive	Uses Windows locale	Supports MBCS
AnsiCompareStr	yes	yes	yes
AnsiCompareText	no	yes	yes
AnsiCompareFileName	no	yes	yes
CompareStr	yes	no	no
CompareText	no	no	no

TABLE Case conversion

Table 3.4 Case conversion routines

Routine	Uses Windows locale	Supports MBCS
AnsiLowerCase	yes	yes
AnsiLowerCaseFileName	yes	yes
AnsiUpperCaseFileName	yes	yes
AnsiUpperCase	yes	yes
LowerCase	no	no
UpperCase	no	no

TABLE Modification

Table 3.5 String modification routines

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
StringReplace	optional by flag	yes
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

TABLE Sub-string

Table 3.6 Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
QuotedStr	no	no

The routines used for string filenames: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the Windows locale. You should always use filenames that are perfectly portable because the locale (character set) used for filenames can and might differ from the default user interface.

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 strings, long strings 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.

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 this section.

String dependencies

Sometimes you will need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. However, 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. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. 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 in a data structure, or return as a value, a local variable. When your routine ends, the *PChar* will disappear because it is simply a pointer to memory, and is 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 that 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 MAXSIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := GetModuleFilename(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. When *GetModuleFilename* returns, the *Length* of the string correctly indicates the number of bytes written to *buf*.

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, 100); // 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.

- `{$H+/-}`: A compiler directive, `$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 `{$H-}` state, and is provided for backwards compatibility with earlier versions of Delphi and Borland Pascal. `$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. Make a link to the compiler directives, since this is a direct quote.
- `{$V+/-}`: The `$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 Delphi's 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.)

Strings and characters: related topics

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see Chapter 10, "Creating international applications."

- "About extended character sets." (Discusses international character sets.)
- "Working with null-terminated strings." (Contains information about character arrays.)
- "Character strings."
- "Character pointers."
- "String operators."

Working with files

This section describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first section discusses the runtime library and Windows API routines you would use

for common programming tasks that involve manipulating files on disk. The next section is an overview of file types used with file I/O. The last section focuses on the recommended approach to working with file I/O, which is to use file streams.

Note Previous versions of the Object Pascal language performed operations on files themselves, rather than on the filename parameters commonly used now. With these older file types you had to locate a file and assign it to a file variable before you could, for example, rename the file.

Manipulating files

There are several common file operations built into Object Pascal's runtime library. The procedures and functions 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. Object Pascal provides routines for most file manipulation. When it does not, alternative routines are discussed.

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 deletions of 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*. In 32-bit Windows you should always use *FindClose* to terminate a *FindFirst/FindNext* sequence. If you want to know if a file exists, there is a *FileExists* function that 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*. The declaration for *TSearchRec* is:

```
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 DOS 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;

```

If a file is found, the fields of the *TSearchRec* type parameter are modified to specify the found file. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Table 3.7 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 with 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*: (*SearchRec.Attr and faHidden > 0*). Attributes can be combined by adding their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly + faHidden*) the *Attr* parameter.

Example: This example 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:

```

var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi4\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)
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
  else
    FindClose(SearchRec);
end;

```

Changing file attributes

Every file has various attributes stored by the operating system as bitmapped flags. File attributes include such items as whether a file is read-only or a hidden file. Changing a file's attributes requires three steps: reading, changing, and setting.

Reading file attributes: Operating systems store file attributes in various ways, generally as bitmapped flags. To read a file's attributes, pass the filename to the *FileGetAttr* function, which returns the file attributes of a file. The return value is a group of bitmapped file attributes, of type *Word*. The attributes can be examined by AND-ing the attributes with the constants defined in *TSearchRec*. A return value of -1 indicates that an error occurred.

Changing individual file attributes: Because Delphi represents file attributes in a set, you can use normal logical operators to manipulate the individual attributes. Each attribute has a mnemonic name defined in the *SysUtils* unit. For example, to set a file's read-only attribute, you would do the following:

```
Attributes := Attributes or faReadOnly;
```

You can also set or clear several attributes at once. For example, the clear both the system-file and hidden attributes:

```
Attributes := Attributes and not (faSysFile or faHidden);
```

Setting file attributes: Delphi enables you to set the attributes for any file at any time. To set a file's attributes, pass the name of the file and the attributes you want to the *FileSetAttr* function. *FileSetAttr* sets the file attributes of a specified file.

You can use the reading and setting operations independently, if you only want to determine a file's attributes, or if you want to set an attribute regardless of previous settings. To change attributes based on their previous settings, however, you need to read the existing attributes, modify them, and write the modified attributes.

Renaming a file

To change a filename, simply use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

which changes a filename, 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, it returns *False*. For example:

```
if not RenameFile('OLDNAME.TXT', 'NEWNAME.TXT') then
  ErrorMessage('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* 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 a Windows 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, take a Windows *Handle* type as a parameter. To get access to a Windows file *Handle* either

- Call the Windows API *CreateFile* function. *CreateFile* is a 32-bit only function that creates or opens a file and returns a *Handle* that can be used to access the file.
- Instantiate *TFileStream* to create or open a file. Then use the *Handle* property as you would a Windows' file *Handle*. See "Using file streams" on page 3-37 for more information.

Copying a file

The runtime library does not provide any routines for copying a file. However, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the Delphi runtime library file routines, *CopyFile* takes a filename as a parameter, not a Window *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 Delphi *RenameFile* function nor the Windows API *MoveFile* function can rename/move files across drives. For more information, see the Microsoft Windows online Help.

File types with file I/O

There are three file types you can use when working with file I/O: Old style Pascal files, Windows file handles, and file stream objects. This section describes these types.

Old style Pascal files: These are the types used with the old file variables, usually of the format "F: Text: or "F: File". There are three classes of these files: typed, text, and untyped and a number of Delphi file-handling routines, such as *AssignPrn* and *writeln*, use them. These file types are obsolete and are incompatible with Windows file handles. If you need to work with the old file types, see the *Object Pascal Language Guide*.

Windows file handles: The Object Pascal file handles are wrappers for the Windows file handle type. The runtime library file-handling routines that use Windows file *Handles* are typically wrappers around Windows API functions. For example, the *FileRead* calls the Windows *ReadFile* function. Because the Delphi functions use Object Pascal syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Using these routines is straightforward, and if you are familiar and comfortable with the Windows API file routines, you may want to use them when working with file I/O.

File streams: File streams are object instances of the VCL *TFileStream* class used to access the information in disk files. File streams are a portable and high level approach to file I/O. *TFileStream* has a *Handle* property that gives you access to the Windows file handle. The next section discusses *TFileStream*.

Using file streams

TFileStream is a VCL class used for high level object representations of file streams. *TFileStream* offers multiple functionality: persistence, interaction with other streams, and file I/O.

- *TFileStream* is a descendant of the stream classes. As such, one advantage of using file streams is that you automatically inherit support for persistence. The stream classes are enabled to work with the *TFile* classes, *TReader* and *TWriter*, to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for the VCL streaming mechanism. For more information about using the VCL streaming system, see the *VCL Reference* online Help on the *TStream*, *TFile*, *TReader*, *TWriter*, and *TComponent* classes.
- *TFileStream* can interact easily with other stream classes. For example, if you want to dump a dynamic memory block to disk, you can do so using a *TFileStream* and a *TMemoryStream*.
- *TFileStream* provides the basic methods and properties for file I/O. The remaining sections focus on this aspect of file streams

Creating and opening files

To create or open a file and get access to a handle for the file, you simply instantiate a *TFileStream*. This opens or creates a named file and provides methods to read from or write to it. If the file can not be opened, *TFileStream* raises an exception.

```
constructor Create(const filename: string; Mode: Word);
```

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:

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 must be one of the following values:

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened.
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.
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

The file open and share mode constants are 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. *Handle* is read-only and indicates 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 Window's 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 Window's file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

Reading and writing to files

TFileStream has several different methods for reading from and writing to files. These are distinguished by whether they perform the following:

- Return the number of bytes read or written.
- Require the number of bytes is known.
- Raise an exception on error.

Read is a function that reads up to *Count* bytes from the file associated with the file stream, starting at the current *Position*, into *Buffer*. *Read* then advances the current position in the file by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint; override;
```

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 end of file marker is encountered.

Write is a function that writes *Count* bytes from the *Buffer* to the file associated with the file stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

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.

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 on error (*EReadError* and *EWriteError*) while the *Read* and *Write* methods do not. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

These methods call the *Read* and *Write* methods, to perform the actual reading and writing.

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 only the pointer element. What you need to do is first cast the string to a *Pointer* or *PChar*, and then dereference it. For example:

```
procedure cast-string;
var
  fs: TFileStream;
  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;
```

Seeking a file

Most typical file I/O mechanisms have a process of seeking a file in order to read from or write to a particular location within it. For this purpose, *TFileStream* provides a *Seek* method. The prototype for *Seek* is:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

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 *Position* of the stream, moving it by the indicated offset. *Seek* returns the new value of the *Position* property, the new current position in the resource.

File position and size

TFileStream has properties that hold the current position and size of the file. These are used by the *Seek*, read, and write methods.

The *Position* property of *TFileStream* is used to indicate the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Longint;
```

The *Size* property indicates the size in bytes of the stream. It is used as an end of file marker to truncate the file. The declaration for *Size* is:

```
property Size: Longint;
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the file. If the *Size* of the file can not be changed, an exception is raised. For example, trying to changes the *Size* for a file that was opened in *fmOpenRead* mode will raise an exception.

Copying

CopyFrom copies a specified number of bytes from one (file) stream to another.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Using *CopyFrom* eliminates the need for the user to create, read into, write from, and free a buffer when copying data.

CopyFrom copies *Count* bytes from *Source* into the stream. *CopyFrom* then moves the current position by *Count* bytes, and returns the number of bytes copied. If *Count* is 0, *CopyFrom* sets *Source* position to 0 before reading and then copies the entire contents of *Source* into the stream. If *Count* is greater than or less than 0, *CopyFrom* reads from the current position in *Source*.

Defining new data types

Object Pascal has many predefined data types. You can use these predefined types to create new types that meet the specific needs of your application. For an overview of types, see the *Object Pascal Language Guide*.

Building applications, components, and libraries

This chapter provides an overview of how to use Delphi to create applications, libraries, and components.

Creating applications

The main use of Delphi is designing and building Windows applications. There are three basic kinds of Windows application:

- Windows GUI applications
- Console applications
- Service applications

Windows applications

When you compile a project, an executable (.EXE) file is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an EXE. You can extend the application by calling DLLs, packages, and other support files from the executable.

Windows 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 multiple document interface (MDI) or single document interface (SDI) form. 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. An SDI application, in contrast, normally contains a single document view. To make your form an SDI application, set the *FormStyle* property of your *Form* object to *fsNormal*.

For more information on developing the UI for an application, see Chapter 5, “Developing the application user interface.”

SDI Applications

To create a new SDI application,

- 1 Select File | New to bring up the New Items dialog.
- 2 Click on the Projects page and select SDI Application.
- 3 Click OK.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so Delphi assumes that all new applications are SDI applications.

MDI applications

To create a new MDI application,

- 1 Select File | New to bring up the New Items dialog.
- 2 Click on the Projects page and select 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 (*fsMDIForm*) or main form (*fsMDIChild*). 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.

Setting IDE, project, and compilation options

Use Project | Project Options to specify various options for your project. For more information, see the online Help.

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 right of the window. All new projects will now have the current options selected by default.

Programming templates

Programming templates are commonly used “skeleton” structures that you can add to your source code and then fill in. For example, if you want to use a **for** loop in your code, you could insert the following template:

```

for := to do
begin

end;

```

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 Select Tools | Environment Options.
- 2 Click the Code Insight tab.
- 3 In the templates section click Add.
- 4 Choose a shortcut name and enter a brief description of the new template.
- 5 Add the template code to the Code text box.
- 6 Click OK.

Console applications

Console applications are 32-bit Windows programs that run without a graphical interface, usually in a console window. These applications typically don't require much user input and perform a limited set of functions.

To create a new console application, choose File | New and select Console Wizard from the New Items dialog box.

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, Choose File | New, and select Service Application from the New Items page. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.

Once you have created a service application, you will see a window in the designer that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the Object Inspector. You can add additional services to your service application by choosing Service from the new items dialog. 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.

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).

Example 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 `WebLogxxx.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 `weblogxxx.log` in the `C:\temp` directory.

- 1 To create the example, choose File | New and select Service Application from the New Items dialog. You will see a window appear named Service1. From the Internet page of the component palette, add a ServerSocket component to the service window (Service1).
- 2 Next, add a private data member of type `TMemoryStream` to the `TService1` class. The interface section of your unit should now look like this:

```
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;
```

- 3 Next, select `ServerSocket 1`, the component you added in step 1. In the Object Inspector, double click the `OnClientRead` event and add the following event handler:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
```

```

begin
  Buffer := nil;
while Socket.ReceiveLength > 0 do begin
  try
    Buffer := AllocMem(Socket.ReceiveLength);
    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;

```

- 4 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:

```

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(False);
    end;

    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;

```

When writing your service application, you should be aware of:

- Service threads
- Service name properties
- Debugging services

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.

Example 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.

- 1 Choose File | New and select Service Application from the New Items dialog. You will see a window appear named Service1.
- 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:

```
TSparkyThread = class(TThread)
public
    procedure Execute; override;
end;
```

- 3 Next, in the implementation section of your unit, create a global variable for a TSparkyThread instance:

```
var
    SparkyThread: TSparkyThread;
```

- 4 Add the following code to the implementation section for the TSparkyThread Execute method (the thread function):

```
procedure TSparkyThread.Execute;
begin
    while not Terminated do
        begin
            Beep;
            Sleep(500);
        end;
end;
```

- 5 Select the Service window (Service1), and double-click the OnStart event in the Object Inspector. Add the following OnStart event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
    SparkyThread := TSparkyThread.Create(False);
    Started := True;
end;
```

- 6 Double-click the OnContinue event in the Object Inspector. Add the following OnContinue event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
    SparkyThread.Resume;
    Continued := True;
end;
```

- 7 Double-click the OnPause event in the Object Inspector. Add the following OnPause event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
    SparkyThread.Suspend;
    Paused := True;
end;
```

- 8 Finally, double-click the OnStop event in the Object Inspector and add the following OnStop event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
    SparkyThread.Terminate;
    Stopped := True;
end;
```

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. These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This section 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 services

Debugging service applications can be tricky, because it requires short time intervals:

- 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:

```
net 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.

Another approach is to attach to the service application process when it is already running. (That is, 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 second 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:

- 1 First create a key called **Image File Execution Options** in the following registry location:

```
HKKEY_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 the debugger as the string value.
- 3 In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

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.

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 Chapter 9, "Working with packages and components."

When to use packages and DLLs

For most applications written in Delphi, 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.

Writing database applications

One of Delphi's strengths is its support for creating advanced database applications. Delphi includes built-in tools that allow you to connect to Oracle, Sybase, Informix, dBASE, Paradox, or other servers while providing transparent data sharing between applications. The Borland Database Engine (BDE) supports scaling from desktop to client/server applications.

Tools, such as the Database Explorer, simplify the task of writing database applications. The Database Explorer is a hierarchical browser for inspecting and modifying database server-specific schema objects including tables, fields, stored procedure definitions, triggers, references, and index descriptions.

Through a persistent connection to a database, Database Explorer lets you

- Create and maintain database aliases
- View schema data in a database, such as tables, stored procedures, and triggers
- View table objects, such as fields and indexes
- Create, view, and modify data in tables
- Enter SQL statements to directly query any database
- Create and maintain data dictionaries to store attribute sets

See Part II, "Developing database applications" in this manual for details on how to use Delphi to create both database client applications and application servers.

Building distributed applications

Distributed applications are applications that are deployed to various machines and platforms and work together, typically over a network, to perform a set of related functions. For instance, an application for purchasing items and tracking those purchases for a nationwide company would require individual client applications for all the outlets, a main server that would process the requests of those clients, and an interface to a database that stores all the information regarding those transactions. By building a distributed client application (for instance, a web-based application), maintaining and updating the individual clients is vastly simplified.

Delphi provides several options for the implementation model of distributed applications:

- TCP/IP applications
- COM and DCOM applications
- CORBA applications
- Database applications

Distributing applications using TCP/IP

TCP/IP is a communication protocol that allows you to write applications that communicate over networks. You can implement virtually any design in your applications. TCP/IP provides a transport layer, but does not impose any particular architecture for creating your distributed application.

The growth of the Internet has created an environment where most computers already have some form of TCP/IP access, which simplifies distributing and setting up the application.

Applications that use TCP/IP can be message-based distributed applications (such as Web server applications that service HTTP request messages) or distributed object applications (such as distributed database applications that communicate using Windows sockets).

The most basic method of adding TCP/IP functionality to your applications is to use client or server sockets. Delphi also provides support for applications that extend Web servers by creating CGI scripts or DLLs. In addition, Delphi provides support for TCP/IP-based database applications.

Using sockets in applications

Two VCL classes, *TClientSocket* and *TServerSocket*, allow you to create TCP/IP socket connections to communicate with other remote applications. For more information on sockets, see Chapter 30, "Working with sockets."

Creating Web server applications

To create a new Web server application, select File | New and select Web Server Application in the New Items dialog box. Then select the Web server application type:

- ISAPI and NSAPI
- CGI stand-alone
- Win-CGI stand-alone

CGI and Win-CGI applications use more system resources on the server, so complex applications are better created as ISAPI or NSAPI applications.

For more information on building Web server applications, see Chapter 29, "Creating Internet server applications".

ISAPI and NSAPI Web server applications

Selecting this type of application sets up your project as a DLL. ISAPI or NSAPI 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.

CGI stand-alone Web server applications

CGI Web server applications are console applications that receive requests from clients on standard input, processes those requests, and sends back the results to the server on standard output to be sent to the client.

Win-CGI stand-alone Web server applications

Win-CGI Web server applications are Windows applications that receive requests from clients from an INI file written by the server and writes the results to a file that the server sends to the client.

Distributing applications using COM and DCOM

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.

COM and DCOM

Delphi has classes and wizards that make it easy to create the essential elements of a COM, OLE, or ActiveX application. 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 Win95 Shell extensions and DirectX multimedia support.

For more information on COM and Active X controls, see Chapter 44, “Overview of COM technologies,” Chapter 48, “Creating an ActiveX control,” and “Distributing a client application as an ActiveX control” on page 14-29.

For more information on DCOM, see “Using DCOM connections” on page 14-8.

MTS

The Microsoft Transaction Server (MTS) is an extension to DCOM that provides transaction services, security, and resource pooling for distributed COM applications.

For more information on MTS, see Chapter 51, “Creating MTS objects,” on page 51-1, “Using MTS” on page 14-5.

Distributing applications using CORBA

Common Object Request Broker Architecture (CORBA) is a method of using distributed objects in applications. The CORBA standard is used on many platforms, so writing CORBA applications allows you to make use of programs that are not running on a Windows machine.

Like COM, CORBA is a distributed object architecture, meaning that client applications can make use of objects that are implemented on a remote server.

For more information on CORBA, see Chapter 28, “Writing CORBA applications.”

For instructions on distributing applications using CORBA, see “Deploying CORBA applications” on page 28-16.

Distributing database applications

Delphi provides support for creating distributed database applications using the MIDAS technology. This powerful technology includes a coordinated set of components that allow you to build a wide variety of multi-tiered database applications. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and OLEnterprise.

For more information about building distributed database applications, see Chapter 14, “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” on page 11-4.

Developing the application user interface

With Delphi, you create a user interface (UI) by selecting components from the Component palette and dropping them onto forms.

Understanding *TApplication*, *TScreen*, and *TForm*

TApplication, *TScreen*, and *TForm* are VCL classes that form the backbone of all Delphi applications by controlling the behavior of your project. The *TApplication* class forms the foundation of a Windows application by providing properties and methods that encapsulate the behavior of a standard Windows program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as system specific information such as screen resolution and what fonts are available for display. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes of your application are based on *TForm*.

Using the main form

TForm is the key class for creating Windows GUI applications.

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 as the project main form and choose OK.

Now if you run the application, your new main form choice is displayed.

Adding additional forms

To add an additional 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).

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 will generate the "Circular reference" error at compile time.

Working at the application level

The global variable *Application*, of type *TApplication*, is in every Delphi Windows application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* would handle how you would 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, 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

An 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, the list of fonts available to the screen device, and multiple screen behavior. If your 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.

Managing layout

At its simplest, you control the layout of your user interface by how 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*:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
    MaxHeight: Integer) of object;
```

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*, *TScrollBar*, *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.

Working with messages

A message is a notification that some event has occurred that is sent by Windows to an application. The message itself is a record passed to a control by Windows. For instance, when you click a mouse button on a dialog box, Windows sends a message to the active control and the application containing that control reacts to this new event. If the click occurs over a button, the *OnClick* event could be activated upon receipt of the message. If the click occurs just in the form, the application can ignore the message.

The record type passed to the application by Windows is called a *TMsg*. Windows predefines a constant for each message, and these values are stored in the message field of the *TMsg* record. Each of these constants begin with the letters *wm*.

The VCL automatically handles messages unless you override the message handling system and create your own message handlers. For more information on messages and message handling, see "Understanding the message-handling system" on page 37-1, "Changing message handling" on page 37-3, and "Creating new message handlers" on page 37-5.

More details on forms

When you create a form in Delphi from the IDE, Delphi automatically creates the form in memory by including code in the *WinMain()* 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 Delphi 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, though, 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 project source unit:

```
Application.CreateForm(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:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    ResultsForm.ShowModal;
end;
```

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, you:

- 1 Select the File | New Form from the Component bar 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 the project source:

```
Application.CreateForm(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.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    ResultsForm:=TResultForm.Create(self)
    ResultsForm.ShowModal;
    ResultsForm.Free;
end;
```

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.

Using a local variable to create a form instance

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:

```

procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;

```

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.

```

TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;

```

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.

```

constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  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;

```

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:

```

procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;

```

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:

```

TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;

```

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*:

```

procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;

```

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:

```

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;

```

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:

```

with ColorForm.ColorListBox do
  MainColor := Items[Index];

```

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 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:

```
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;
```

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:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

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” on page 5-5.

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:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
    end;
  Close;
end;
```

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:

```

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;

```

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

Delphi offers several ways to save and reuse work you've done with VCL components:

- *Component templates* provide a simple, quick way of configuring and saving groups of components. See "Creating and using component templates" on page 5-12.
- You can save forms, data modules, and projects in the *Repository*. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes. See "Using the Object Repository" on page 2-35.
- You can save *frames* on the Component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames. See "Working with frames" on page 5-12.
- Creating a *custom component* is the most complicated way of reusing code, but it offers the greatest flexibility. See Chapter 31, "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 Component 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 Name 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 Component 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 Under 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 Component 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 Component 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.

Creating frames

To create an empty frame, choose File | New Frame, or choose File | New and double-click on Frame. You can now 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 .DFM file by right-clicking and choosing View as Form or View as Text.

Adding frames to the Component palette

Frames are added to the Component palette as component templates. To add a frame to the Component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click on 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:

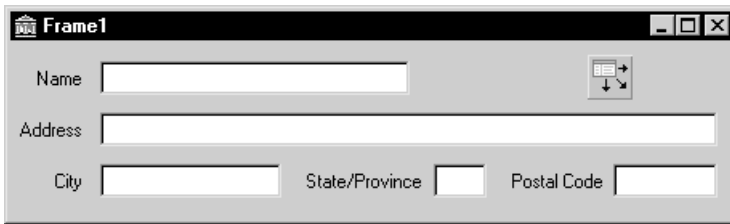
- Select a frame from the Component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.
- Select *Frames* from the Standard page of the Component palette and click on a form 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.

Figure 5.1 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) 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" on page 2-35.

If you send a frame's unit and form files to other developers, they can open them and add them to the Component palette. If the frame has other frames embedded in it, they will have to open it as part of a project.

Creating and managing menus

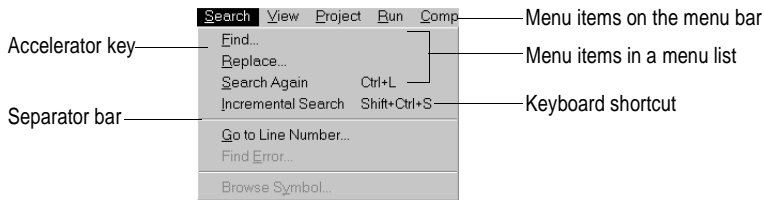
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predefined or custom tailored—to your form. You simply add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer 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 chapter 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

Figure 5.2 Menu terminology

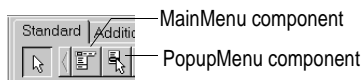


For information about hooking up menu items to the code that executes when they are selected, see “Associating menu events with event handlers” on page 2-26.

Opening the Menu Designer

To start using the Menu Designer, first add either a MainMenu or PopupMenu component to your form. Both menu components are located on the Standard page of the Component palette.

Figure 5.3 MainMenu and PopupMenu components



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 choose from one of the following methods:

- Double-click the menu component.
- 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 highlighted in the Object Inspector.

Figure 5.4 Menu Designer for a main menu

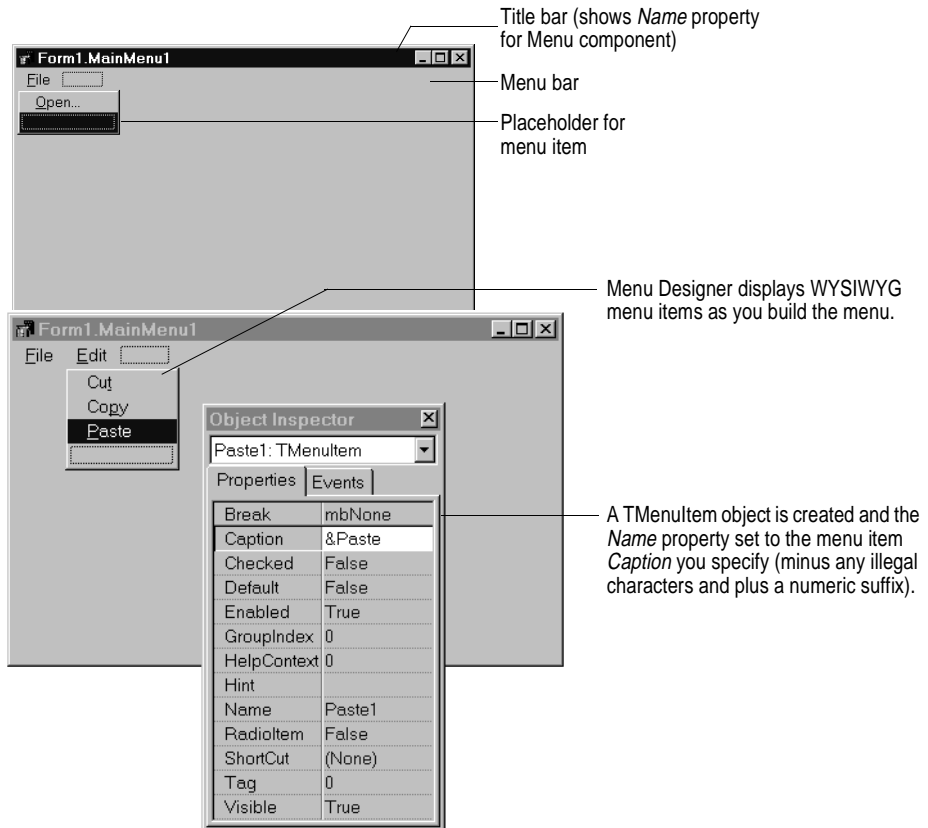


Figure 5.5 Menu Designer for a pop-up menu

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.

This section discusses the basics of creating a menu at design time. For more information about menu templates, see “Using menu templates” on page 5-24.

Naming menus

As with all components, when you add a menu component to the form, Delphi gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Delphi adds the menu name 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 in the value for the *Name* property.
- Type in 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 in a value.

Note

If you enter characters in the *Caption* property that are not valid for Object Pascal 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.

Table 5.1 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.

If you've just opened the Menu Designer, the first position on the menu bar is already selected.

- 2 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" on page 5-17.)

- 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.

Adding separator bars

Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

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 accessing 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.

Caution Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

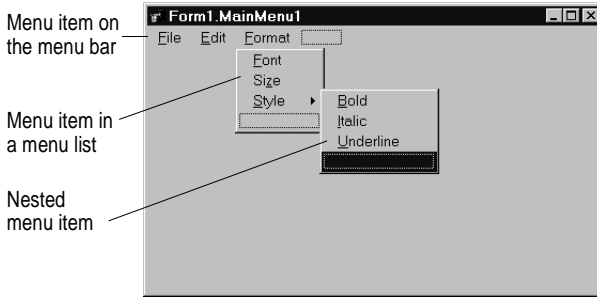
Creating submenus

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.)

Figure 5.6 Nested menu structures



To create a submenu,

- 1 Select the menu item under which you want to create a submenu.
- 2 Press *Ctrl*→ 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 ↓, 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. To add an image to a menu item:

- 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 above.
- 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 section 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.

For information about assigning event handlers to menus, see “Associating menu events with event handlers” on page 2-26.

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” on page 5-24.)

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.

Table 5.2 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 an .MNU file to open in the current form.

Switching between menus at design time

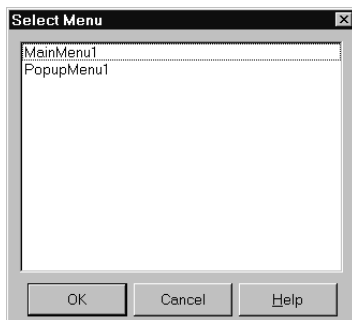
If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

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.

Figure 5.7 Select Menu dialog box



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

Delphi provides several predesigned menus, or menu templates, that 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 shipped with Delphi are stored in the BIN subdirectory in a default installation. These files have a .DMT (Delphi menu template) 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.

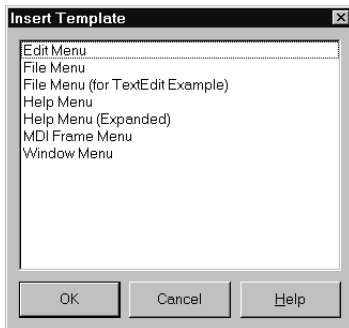
To add a menu template to your application,

- 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.

Figure 5.8 Sample Insert Template dialog box for menu



- 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

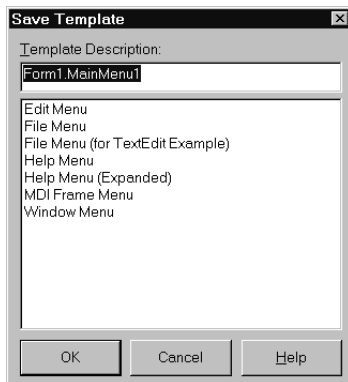
Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

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.

Figure 5.9 Save Template dialog box for menus



- 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 an event with an existing event handler” on page 2-25.

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” on page 6-9.

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. The following section discusses this in more detail.

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*.

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,

```
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*:

- 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

Delphi supports menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi 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.

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.

There are several ways to add a toolbar to a form:

- 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

- Add a toolbar and corresponding speed button controls using the panel component
- Add a toolbar and corresponding tool button controls using the Toolbar component
- Add a cool bar using the cool bar component
- Respond to clicks
- Add hidden toolbars and cool bars
- Hide and show toolbars and cool bars

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 page of the Component 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:

- Add a speed button to a toolbar panel
- Assign a speed button's glyph
- Set the initial condition of a speed button
- Create a group of speed buttons
- Allow 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 page of the Component 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.

Table 5.3 lists some actions you can set to change a speed button's appearance:

Table 5.3 Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	<i>GroupIndex</i> property to a value other than zero and its <i>Down</i> property to <i>True</i> .
Appear disabled	<i>Enabled</i> property to <i>False</i> .
Have a left margin	<i>Indent</i> 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 page of the Component 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:

- Add a tool button
- Assign images to tool buttons
- Set the tool buttons' appearance
- Create a group of tool buttons
- Allow 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 Component 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

Table 5.4 lists some actions you can set to change a tool button's appearance:

Table 5.4 Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	<i>GroupIndex</i> property to a nonzero value and its <i>Down</i> property to <i>True</i> .
Appear disabled	<i>Enabled</i> property to <i>False</i> .
Have a left margin	<i>Indent</i> property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	<i>Flat</i> property to <i>True</i> .

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

The cool bar component—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,

- 1 Add a cool bar component to the form (from the Win32 page of the Component palette). The cool bar automatically aligns to the top of the form.
- 2 Add windowed controls from the Component palette to the bar.

Only components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to the cool bar, but they will not appear on separate bands.

Note The cool bar component requires version 4.70 or later of COMCTL.DLL.

Setting the appearance of the cool bar

The cool bar component offers several useful configuration options. Table 5.5 lists some actions you can set to change a tool button's appearance:

Table 5.5 Setting a cool button's appearance

To make the cool bar:	Set the toolbar's:
Resize automatically to accommodate the bands it contains	<i>AutoSize</i> property to <i>True</i> .
Bands maintain a uniform height	<i>FixedSize</i> property to <i>True</i> .
Reorient to vertical rather than horizontal	<i>Vertical</i> property to <i>True</i> . This changes the effect of the <i>FixedSize</i> property.
Prevent the <i>Text</i> properties of the bands from displaying at runtime	<i>ShowText</i> property to <i>False</i> . Each band in a cool bar has its own <i>Text</i> property.
Remove the border around the bar	<i>BandBorderStyle</i> to <i>bsNone</i> .
Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)	<i>FixedOrder</i> to <i>True</i> .
Create a background image for the cool bar	<i>Bitmap</i> property to <i>TBitmap</i> object.
Choose a list of images to appear on the left of any band	<i>Images</i> property to <i>TImageList</i> 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.

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 more information, see “Working with events and event handlers” on page 2-24 and “Generating a handler for a component’s default event” on page 2-25.

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:

- 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 hide or show 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.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
    PenBar.Visible := PenButton.Down;
end;
```

Using action lists

Action lists let you centralize the response to user commands (actions) for objects such as menus and buttons that respond to those commands. This section is an overview of actions and action lists, describing how to use them and how they interact with their clients and targets.

Action objects

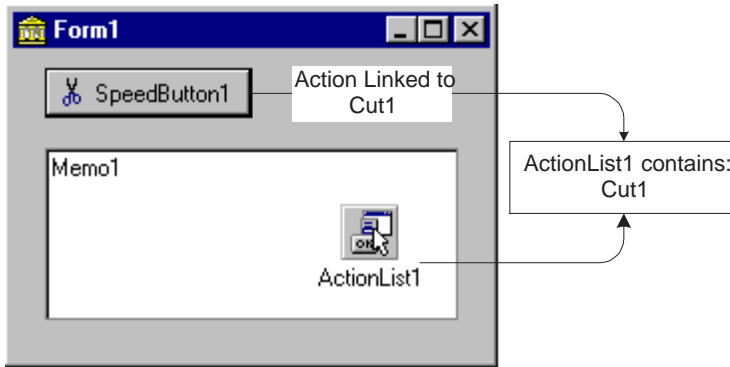
Actions are user commands that operate on target objects. You create actions in the action list component editor. These actions are later connected to client controls via their action links. Following are descriptions of each type of component in the action/action list mechanism:

- An action (*TAction*) is the implementation of an action, such as copying highlighted text, on a target, such as an edit control. An action is triggered by a client in response to a user command (such as a mouse click). Clients are typically menu items or buttons. The *StdActns* unit contains classes derived from *TAction* that implement the basic Edit and Window menu commands (actions) found in most Window applications.
- An action list (*TActionList*) is a component that maintains a list of actions (*TAction*). Action lists are the design-time user interface for working with actions.
- An action link (*TActionLink*) is an object that maintains the connection between actions and clients. Action links determine whether an action, or which action, is currently applicable for a given client.
- A client of an action is typically a menu item or a button (*TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton*, and so on). An action is initiated by a corresponding command in the client. Typically a client *Click* is associated with an action *Execute*.
- An action target is usually a control, such as a rich edit, a memo, or a data control. The *DBActns* unit, for example, contains classes that implement actions specific to data set controls. Component writers can create their own actions specific to the needs of the controls they design and use, and then package those units to create more modular applications.

Figure 5.10 shows the relationship of these objects. In this diagram, *Cut1* is the action, *ActionList1* is the action list containing *Cut1*, *SpeedButton1* is the client of *Cut1*, and *Memo1* is the target. Unlike actions, action lists, action clients, and action targets, action links are non-visual objects. The action link in this diagram is therefore

indicated by a white rectangle. The action link associates the *SpeedButton1* client to the *Cut1* action contained in *ActionList1*.

Figure 5.10 Action list mechanism



The VCL includes *TAction*, *TActionList*, and *TActionLink* type classes for working with Action lists. By unit, these are

- ActnList.pas: *TAction*, *TActionLink*, *TActionList*, *TContainedAction*, *TCustomAction*, and *TCustomActionList*
- Classes.pas: *TBasicAction* and *TBasicActionLink*
- Controls.pas: *TControlActionLink* and *TWinControlActionLink*
- ComCtrls.pas: *TToolButtonActionLink*
- Menus.pas: *TMenuActionLink*
- StdCtrls.pas: *TButtonActionLink*

There are also two units, *StdActns* and *DBActns*, that contain auxiliary classes that implement specific, commonly used standard Windows and data set actions. These are described in “Pre-defined action classes” on page 5-39. Many of the VCL controls include properties (such as *Action*) and methods (such as *ExecuteAction*) that enable them to be used as action clients and targets.

Using actions

You can add an action list to your forms or data modules from the standard page of the Component Palette. Double-click the action list to display the Action List editor, which lets you add, delete, and rearrange actions in much the same way you use the collection editor.

In the Object Inspector, set the properties for each action. The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, and *Visible*) correspond to the properties of client controls. These are typically, but not necessarily, the same name as the client property. For example, an action’s *Checked* property corresponds to a *TToolButton*’s *Down* property.

Centralizing code

A number of controls such as *TToolButton*, *TSpeedButton*, *TMenuItem*, and *TButton* have a published property called *Action*. When you set the *Action* property to one of the actions in your action list, the values of the corresponding properties in the action are copied to those of the control. All properties and events in common with the action object (except *Name* and *Tag*) are dynamically linked to the control. Thus, for example, instead of duplicating the code that disables buttons and menu items, you can centralize this code in an action object, and when the action is disabled, all corresponding buttons and menu items are disabled.

Linking properties

The client's action link is the mechanism through which its properties are associated with (linked to) the properties of an action. When an action changes, the action link is responsible for updating the client's properties. For details about which properties a particular action link class handles, refer to the individual action link classes in the VCL reference.

You can selectively override the values of the properties controlled by an associated action object by setting the property's value in the client component or control. This does not change the property in the action, so only the client is affected.

Executing actions

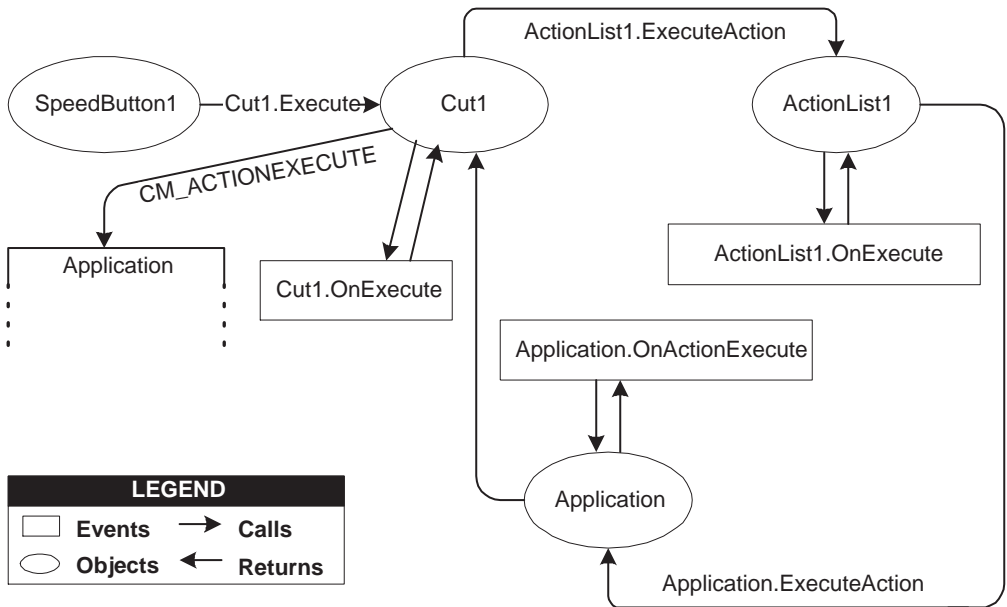
When a client component or control is clicked, the *OnExecute* event occurs for its associated action. For example, the following code illustrates the *OnExecute* event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
    { Toggle Toolbar1's visibility }
    Toolbar1.Visible := not Toolbar1.Visible;
end;
```

Note If you are using a tool button or a menu item, you must manually set the *Images* property of the corresponding toolbar or menu component to the *Images* property of the action list. This is true even though the *ImageIndex* property is dynamically linked to the client.

For general information about events and event handlers, see “Working with events and event handlers” on page 2-24.

Figure 5.11 illustrates the dispatching sequence for the execution cycle of an action called *Cut1*. This diagram assumes the relationship of the components in Figure 5.10, meaning that the *Speedbutton1* client is linked to the *Cut1* action via its action link. *Speedbutton1*'s *Action* property is therefore *Cut1*. Consequently, *Speedbutton1*'s *Click* method invokes *Cut1*'s *Execute* method.

Figure 5.11 Execution cycle for an action

Note In the description of this sequence, one method invoking another does not necessarily mean that the invocation is explicit in the code for that method.

Clicking on *Speedbutton1* initiates the following execution cycle:

- *Speedbutton1*'s *Click* method invokes *Cut1.Execute*.
- The *Cut1* action defers to its action list (*ActionList1*) for the processing of its *Execute*. This is done by calling the Action list's *ExecuteAction* method, passing itself as a parameter.
- *ActionList1* calls its event handler (*OnExecute*) for *ExecuteAction*. (An action list's *ExecuteAction* method applies to all actions contained by the action list.) This handler has a parameter *Handled*, that returns *False* by default. If the handler is assigned and handles the event, it should return *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    { Prevent execution of actions contained by ActionList1 }
    Handled := True;
end;
```

If execution is not handled, at this point, in the action list event handler, then processing continues:

- The *Cut1* action is routed to the *Application* object's *ExecuteAction* method, which invokes the *OnActionExecute* event handler. (The application's *ExecuteAction* method applies to all of the actions in that application.) The sequence is the same as for the action list *ExecuteAction*: The handler has a parameter *Handled* that

returns *False* by default. If the handler is assigned and handles the event, it should return *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    { Prevent execution of all actions in Application }
    Handled := True;
end;
```

If execution is not handled in the application's event handler, then *Cut1* send the *CM_ACTIONEXECUTE* message to the application's *WndProc*, passing itself as a parameter. The application then tries to find a target on which to execute the action (see Figure 5.12, "Action targets").

Updating actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a visible 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:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
    { Indicate whether ToolBar1 is currently visible }
    (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

See also the RichEdit demo(Demos\RichEdit).

The dispatching cycle for updating actions follows the same sequence as the execution cycle in Figure 5.11.

Note 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.

Pre-defined action classes

Component writers can use the classes in the *StdActns* and *DBActns* units as examples for deriving their own action classes that implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*) 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.

Standard edit actions

The standard edit actions are designed to be 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 Windows Clipboard.

- *TEditAction* ensures that the target control is a *TCustomEdit* class (or descendant).

- *TEditCopy* copies highlighted text to the Clipboard.
- *TEditCut* cuts highlighted text from the target to the Clipboard.
- *TEditPaste* pastes text from the Clipboard to the target and ensures that the Clipboard is enabled for the text format.
- *TEditDelete* deletes the highlighted text.
- *TEditSelectAll* selects all the text in the target edit control.
- *TEditUndo* undoes the last edit made to the target edit control.

Standard Window actions

The standard Window actions are designed to be 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.

- *TWindowAction* ensures that the target control is a *TForm* class and checks whether the form has MDI child forms.
- *TWindowArrange* arranges the icons of minimized MDI child forms.
- *TWindowCascade* cascades the MDI child forms.
- *TWindowClose* closes the active MDI child form.
- *TWindowMinimizeAll* minimizes all of the MDI child forms.
- *TWindowTileHorizontal* arranges MDI child forms so that they are all the same size, tiled horizontally.
- *TWindowTileVertical* arranges MDI child forms so that they are all the same size, tiled vertically.

Standard Help actions

The standard Help actions are designed to be used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command on to WinHelp.

- *THelpAction* ensures that the global *Application* variable is available, so that commands can be handled using its *HelpCommand* method.
- *THelpContents* brings up the Help Topics dialog on the tab (Contents, Index or Find) that was last used.
- *THelpTopicSearch* brings up the Help Topics dialog on the Index tab.
- *THelpOnHelp* brings up the Microsoft help file on how to use Help. Note that this file is an HTML help file on recent versions of Windows, and does not describe the WinHelp system.

DataSet actions

The standard dataset actions are designed to be 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.

The *TDataSetAction* introduces a *DataSource* property which ensures actions are performed on that dataset. If *DataSource* is **nil**, the currently focused data-aware control is used. For details, refer to Figure 5.12, “Action targets,” on page 5-42.

- *TDataSetAction* ensures that the target is a *TDataSource* class and has an associated data set.
- *TDataSetCancel* cancels the edits to the current record, restores the record display to its condition prior to editing, and turns off Insert and Edit states if they are active.
- *TDataSetDelete* deletes the current record and makes the next record the current record.
- *TDataSetEdit* puts the dataset into *Edit* state so that the current record can be modified.
- *TDataSetFirst* sets the current record to the first record in the dataset.
- *TDataSetInsert* inserts a new record before the current record, and sets the dataset into Insert and Edit states.
- *TDataSetLast* sets the current record to the last record in the dataset.
- *TDataSetNext* sets the current record to the next record.
- *TDataSetPost* writes changes in the current record to the dataset.
- *TDataSetPrior* sets the current record to the previous record.
- *TDataSetRefresh* refreshes the buffered data in the associated dataset.

Writing action components

The pre-defined actions are examples of extending the VCL action classes. The following topics are useful if you are writing your own action classes:

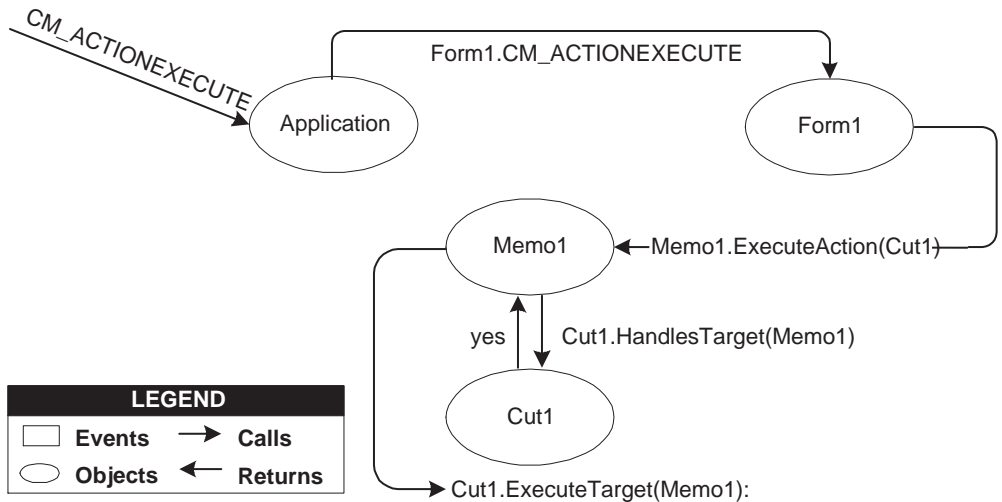
- How actions find their targets
- Registering actions
- Writing action list editors

How actions find their targets

Figure 5.11 illustrates the execution cycle for the standard VCL action classes. If execution is not handled by the action list, the application, or the default action event handlers, then the `CM_ACTIONEXECUTE` message is sent to the application’s *WndProc*. Figure 5.12 continues the execution sequence at this point. The pre-defined

action classes described previously as well as any action class that you create, use this path of execution:

Figure 5.12 Action targets



- Upon receiving the `CM_ACTIONEXECUTE` message the application first dispatches it to the Screen's *ActiveForm*. If there is no active form, the application sends the message to its *MainForm*.
- *Form1* (in this example, the active form) first looks for the active control (*Memo1*) and calls that control's *ExecuteAction* method passing *Cut1* as a parameter.
- *Memo1* calls *Cut1*'s *HandlesTarget* method, passing itself to determine whether it is an appropriate target for the action. If *Memo1* is not an appropriate target, *HandlesTarget* returns *False* and *Memo1*'s *ExecuteAction* handler returns *False*.
- In this case, *Memo1* is an appropriate target for *Cut1*, so *HandlesTarget* returns *True*. *Memo1* then calls *Cut1.ExecuteTarget* passing itself as a parameter. Finally, since *Cut1* is an instance of a *TEditCut* action, the action calls *Memo1*'s *CutToClipboard* method:

```
procedure TEditCut.ExecuteTarget(Target: TObject);
begin
    GetControl(Target).CutToClipboard;
end;
```

If the control were not an appropriate target, processing would continue as follows:

- *Form1* calls its *ExecuteAction* method. If *Form1* is an appropriate target (for example, a form would be a target for the *TWindowCascade* action) then it calls *Cut1*'s *ExecuteTarget* method, passing itself as a parameter.
- If *Form1* is not an appropriate target, it invokes *ExecuteAction* on every visible control it owns until a target is found.

Note If the action involved is a *TCustomAction* type, then the action is automatically disabled for you if, the action is not handled and, its *DisableIfNoHandler* property is *True*.

Registering actions

You can register and unregister your own actions with the IDE by using the global routines in the *ActnList* unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
  TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

Use these routines the same way you would when registering components (*RegisterComponents*). For example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
  TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

Writing action list editors

You may want to write your own component editor for action lists. If you do, you can assign your own procedures to the four global procedure variables in the *ActnList* unit:

```
CreateActionProc: function (AOwner: TComponent; ActionClass: TBasicActionClass):
  TBasicAction = nil;

EnumRegisteredActionsProc: procedure (Proc: TEnumActionProc; Info: Pointer) = nil;

RegisterActionsProc: procedure (const CategoryName: string; const AClasses: array of
  TBasicActionClass; Resource: TComponentClass) = nil;

UnRegisterActionsProc: procedure (const AClasses: array of TBasicActionClass) = nil;
```

You only need to reassign these if you want to manage the registration, unregistration, creation, and enumeration procedures of actions differently from the default behavior. If you do, write your own handlers and assign them to these variables within the initialization section of your design-time unit.

Demo programs

For examples of programs that use actions and action lists, refer to *Demos\RichEdit*. In addition, the Application wizard (File | New Project page) demos, MDI Application, SDI Application, and Win95 Logo Application can use the action and action list objects.

Working with controls

Controls are visual components that the user can interact with at runtime. This chapter describes a variety of features common to many 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*. If you pass *True*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse a short distance. Calling `BeginDrag(False)` 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.

```

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;
end;

```

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. If *Accept* is *True*, the application sends a drag-drop event to the control.

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 drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

```

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;

```

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-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 as they are passed over.

In the following 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.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileList.FileName, Items[GetItem(X, Y)].FullPath);
    end;
end;
```

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 item was dragged. 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 this example, a file list box handles an end-drag event by refreshing its file list.

```
procedure TFMForm.FileList1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileList1.Update;
end;
```

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-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* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-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-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

Drag objects let you drag items between a form implemented in the application's main EXE file and a form implemented in a DLL, or between forms that are implemented in 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.

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

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.

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.

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:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

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*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer; State:
TDragState; var Accept: Boolean) of object;
```

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.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer) of
object;
```

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.

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:

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean) of object;
```

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.

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.

Working with text in controls

The following sections explain 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 the `Character | Left` menu item, then attaches the same event handler to both the `Right` and `Center` menu items on the `Character` menu.

```

procedure TEditForm.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;

```

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.

```

procedure TEditForm.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;
end;

```

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. The *Clipboard* object in Delphi encapsulates 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

```
uses Clipbrd;
```

For example, in an application with a child window, the `uses` clause in the unit's `implementation` part might look like this:

```
uses
  MDIFrame, Clipbrd;
```

Selecting text

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.

Table 6.1 lists properties commonly used to handle selected text.

Table 6.1 Properties of selected text

Property	Description
<i>SelText</i>	Contains a string representing the selected text in the component.
<i>SelLength</i>	Contains the length of a selected string.
<i>SelStart</i>	Contains the starting position of a string.

Selecting all text

The *SelectAll* method selects the entire contents of the 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,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll; { select all text in RichEdit }
end;
```

Cutting, copying, and pasting text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the Windows Clipboard. The edit components that encapsulate the standard Windows text-handling controls all have methods built into them for interacting with the Clipboard. (See “Using the Clipboard with graphics” on page 7-20 for information on using the Clipboard with graphics.)

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:

```
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;
```

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:

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

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.

```
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;
```

The *HasFormat* method 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.

Chapter 7, “Working with graphics and multimedia” provides more information about using the Clipboard with graphics.

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, as described in “Disabling menu items” on page 6-9.

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 *EditEditClick* event handler described previously in “Disabling menu items” on page 6-9 is attached to the pop-up menu component’s *OnPopup* event. A line of code is added to *EditEditClick* for each item in the pop-up menu.

```
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}
  Deletel.Enabled := HasSelection;
end;
```

Adding graphics to controls

Windows list-box, combo-box, and menu controls have a style available called “owner draw,” which means that instead of using Windows’ standard method of drawing text for each item in the control, the control’s owner (generally, the form) draws each item 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” on page 5-21.

All owner-draw controls contain lists of items. By default, those lists are lists of strings, which Windows displays as text. You can associate an object with each item in a list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Delphi involves these steps:

- 1 Setting the owner-draw style
- 2 Adding graphical objects to a string list
- 3 Drawing owner-drawn items

Setting the owner-draw style

Both list boxes and combo boxes 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 boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as Table 6.2 describes. Owner-draw grids are always fixed: although the size of each row and column might vary, the size of each cell is determined before drawing the grid.

Table 6.2 Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the <i>ItemHeight</i> property.	<i>lbOwnerDrawFixed</i> , <i>csOwnerDrawFixed</i>
Variable	Each item might have a different height, determined by the data at runtime.	<i>lbOwnerDrawVariable</i> , <i>csOwnerDrawVariable</i>

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.

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.

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, you can store bitmaps for owner-draw controls in hidden image controls, like this:

- 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.

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:

```

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;

```

Drawing owner-drawn items

When you set a control's style to owner draw, Windows no longer draws the control on the screen. Instead, it 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 *lsOwnerDrawFixed*), 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, Windows generates a measure-item event. The measure-item event tells the application where the item appears on the control.

Windows determines the size 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 Windows chose. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to be the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be big 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 size of that item. The size 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.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer); { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

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 each owner-draw item

When an application needs to draw or redraw an owner-draw control, Windows generates draw-item events for each visible item in the control.

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 always start with *OnDraw*, such as *OnDrawItem* or *OnDrawCell*.

The draw-item event contains parameters indicating the index of 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:

```

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;

```


Working with graphics and multimedia

Graphics and multimedia elements can add polish to your applications. Delphi offers a variety of ways to introduce these features into your application. 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, Delphi includes special components that can play audio and video clips.

Overview of graphics programming

The VCL graphics components encapsulate the Windows Graphics Device Interface (GDI), making it very easy to add graphics to your Windows programming.

To draw graphics in a Delphi 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 takes care of device context, 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. 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.

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.

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 examples in the beginning of this chapter demonstrate how to draw various graphics, but they do so in response to *OnPaint* events. Later sections show how to do the same kind of drawing in response to other events.

Refreshing the screen

At certain times, Windows determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages, which the VCL routes to *OnPaint* events. The VCL calls any *OnPaint* event handler that you have written for that object 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 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, the painting and refreshing of the graphic contained in the *TImage* is handled automatically by the VCL. 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, 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 image to be repainted each time the client area is invalidated.

Types of graphic objects

The VCL provides the graphic objects shown in Table 7.1. These objects have methods to draw on the canvas, which are described in "Using Canvas methods to

draw graphic objects” on page 7-9 and to load and save to graphics files, as described in “Loading and saving graphics files” on page 7-18.

Table 7.1 Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the <i>Picture Register</i> 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 <i>handle</i> 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 a Windows icon file (:ICO file).
Metafile	Contains a metafile, which records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles 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 do not draw as fast as bitmaps. Use a metafile when versatility or precision is more important than performance.

Common properties and methods of Canvas

Table 7.2 lists the commonly used properties of the Canvas object. For a complete list of properties and methods, see the *TCanvas* component in online Help.

Table 7.2 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 <i>TFont</i> 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 <i>TBrush</i> 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 <i>TPen</i> 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 <i>ClipRect</i> .

These properties are described in more detail in “Using the properties of the Canvas object” on page 7-5.

Table 7.3 is a list of several methods you can use:

Table 7.3 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	Fills an area of the canvas using the current brush.
FrameRect	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 <i>Rectangle</i> 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” on page 7-9.

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 section 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*, *Width*, *Style*, and *Mode*.

- 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.

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:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

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 95 always draw solid lines, no matter what 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:

```
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;
```

Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

Note Windows 95 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*.

Delphi 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:

```
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;
```

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. See *TPen* in online Help for details.

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:

```
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.

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 (see "Changing the pen color" on page 7-5):

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

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: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, or *bsDiagCross*.

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:

```
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;
```

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:

```
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;
```


Note The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remain 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 section 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:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    MoveTo(0, 0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

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.

The following method, for example, draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
    PolyLine([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
  end;
end;
```

This example takes advantage of Delphi's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter. For more information, see online Help.

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 section covers:

- 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.

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

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.

The following method, for example, draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

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.

For example, the following code draws a right triangle in the lower left half of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
    Point(ClientWidth, ClientHeight)]);
end;
```

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
- Changing the tool with speed buttons
- Using 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, Object Pascal 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 Object Pascal'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:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

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:

```
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 Object Pascal'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:

```
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;
```

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:

```
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;

```

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 section 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:

```

procedure TForm1.FormMouseUp(Sender: TObject);
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;

```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```

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;

```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section 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.

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers. First, the declaration of *DrawShape* is added to the form object's declaration:

```

type
  TForm1 = class(TForm)
    ...{ fields and methods declared here}
  public
    { Public declarations }
    procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;

```

Then, the implementation of *DrawShape* is written in the implementation part of the unit:

```

implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
    begin
      Pen.Mode := AMode;
      case DrawingTool of
        dtLine:
          begin
            MoveTo(TopLeft.X, TopLeft.Y);
            LineTo(BottomRight.X, BottomRight.Y);
          end;
        dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
        dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
          (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
      end;
    end;
  end;
end;

```

The other event handlers are modified to call *DrawShape*.

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
  Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
    begin
      DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
      MovePt := Point(X, Y);{ record the current point }
      DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
    end;
end;
end;

```

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 a VCL 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, metafiles, 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. Contrastly, 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" on page 6-11 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” on page 7-18.

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:

```

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 }
end;

```

Assigning the bitmap to the picture’s *Graphic* property gives ownership of the bitmap to the picture object. The picture object destroys the bitmap when it finishes with it, so you should not destroy the bitmap object. You can assign a different bitmap to the picture (see “Replacing the picture” on page 7-19), at which point the picture disposes of the old bitmap and assumes ownership of the new one.

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.

```

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;

```

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 VCL's image control makes it easy to load pictures from a file and save them again.

The VCL components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, 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 the following sections:

- 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-file dialog box, and then loads that file into an image control named *Image*:

```

procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenFileDialog1.Execute then
  begin
    CurrentFile := OpenFileDialog1.FileName;
    Image.Picture.LoadFromFile(CurrentFile);
  end;
end;

```

Saving a picture to a file

The VCL 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 section.

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.

```

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;

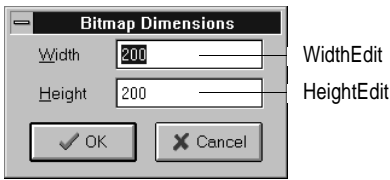
```

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 (see “Setting the initial bitmap size” on page 7-16), 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, such as the one in Figure 7.1.

Figure 7.1 Bitmap-dimension dialog box from the BMPDIg unit.

This particular dialog box is created in the *BMPDIg* unit included with the *GraphEx* project (in the `EXAMPLES\DOC\GRAPHEX` directory).

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:

```

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 }
                end;
            end;
        end;
end;

```

Note Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to destroy the existing bitmap and take ownership of the new one. The VCL handles the details of freeing the resources associated with the previous bitmap automatically.

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 Clipboard unit to the uses clause of any unit that needs to access Clipboard data.

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 Windows 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:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
    Clipboard.Assign(Image.Picture)
end.
```

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:

```
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;
```

Pasting graphics from the Clipboard

If the Windows 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 to see whether the Clipboard contains a graphic.

HasFormat is a Boolean function. It returns *True* if the Clipboard contains an item of the type specified in the parameter. To test for graphics, you pass *CF_BITMAP*.

- 2 Assign the Clipboard to the destination.

This 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:

```

procedure TForm1.PasteButtonClick(Sender: TObject);
var
    Bitmap: TBitmap;
begin
    if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Clipboard? }
    begin
        Image.Picture.Bitmap.Assign(Clipboard);
    end;
end;

```

The graphic on the Clipboard could come from this application, or it could have been copied from another application, such as Windows Paintbrush. 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 section walks you through 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 in this section is taken from a sample application located in the EXAMPLES\DOC\GRAPHEX directory. 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.

This section covers:

- 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 section covers:

- 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?

The VCL has three mouse events: *OnMouseDown* event, *OnMouseMove* event, and *OnMouseUp* event.

When a VCL 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:

Table 7.4 Mouse-event parameters

Parameter	Meaning
<i>Sender</i>	The object that detected the mouse action
<i>Button</i>	Indicates which mouse button was involved: <i>mbLeft</i> , <i>mbMiddle</i> , or <i>mbRight</i>
<i>Shift</i>	Indicates the state of the <i>Alt</i> , <i>Ctrl</i> , and <i>Shift</i> keys at the time of the mouse action
<i>X, Y</i>	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 VCL generates an empty handler for a mouse-down event on the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
end;
```

Here's code that displays some text at the point where the mouse button is pressed. It uses the *X* and *Y* parameters sent to the method, and calls the *TextOut* method of the canvas to display text there:

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.TextOut(X, Y, 'Here!'); { write text at (X, Y) }
end;
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, “Here!” appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(X, Y);{ set pen position }
end;
```

Pressing the mouse button now sets the pen position, setting the line’s starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

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.

Here’s a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

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:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```


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.

The following code gives a form a field called *Drawing* of type Boolean, in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of type TPoint.

```

type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  public
    Drawing: Boolean; { field to track whether button was pressed }
    Origin, MovePt: TPoint; { fields to store points }
  end;

```

When you have a *Drawing* field to track whether to draw, set it to *True* when the user presses the mouse button, and *False* when the user releases it:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True; { set the Drawing flag }
  Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False; { clear the Drawing flag }
end;

```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is *True*:

```

procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
    Canvas.LineTo(X, Y);
end;

```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

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:

```

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;

```

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:

```

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;

```

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:

```

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;

```

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

Delphi allows you to add multimedia components to your applications. To do this, you can use either the *TAnimate* component on the Win32 page or the *TMediaPlayer* component on the System page of the Component 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.

For more information on *TAnimate* and *TMediaPlayer* components, see the VCL online help.

The following topics are discussed in this section:

- Adding silent video clips to an application
- Adding audio and/or video clips to an application

Adding silent video clips to an application

The animation control in Delphi allows you to add silent video clips to your application.

To add a silent video clip to an application:

- 1 Double-click the animate icon on the Win32 page of the Component 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 *name* 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 *Common AVI* property and choose one of the AVIs available from the drop down list; or
 - Select the *FileName* property and click the ellipsis (...) button, choose an AVI file from any available local or network directories and click Open in the Open AVI dialog; 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*.

This loads the AVI file into memory. If you want to display the first frame of the AVI 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 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 *StartFrame* property 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 clip when a specific event takes place at runtime. For example, to activate the AVI 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.

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.

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 page of the Component palette.
- 2 Using the Object Inspector, set its Name property to *Logo1*.
- 3 Select its *FileName* property, click the ellipsis (...) button, choose the cool.avi file from your ..\Demos\Coolstuf directory. Then click Open in the Open AVI dialog.

This loads the cool.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:

```
Logo1.Active := True;
```

- 8 Double-click the Label icon on the Standard page of the Component 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:

```
LogoForm1.Close;
```

- 10 Select Run | Run to execute the animated logo window.

Adding audio and/or video clips to an application

The media player component in Delphi allows you to 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.

To add an audio and/or video clip to an application:

- 1 Double-click the media player icon on the System page of the Component 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 Table 7.5.
- 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 mediaplayer 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*, *Start Recording*, *Eject*).

- 8 Make any other changes to the media player 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.

Table 7.5 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

Example of adding audio and/or video clips

This example runs an AVI video clip of a multimedia advertisement for Delphi. 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 page of the Component 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, choose the speeds.avi file from your ..\Demos\Coolstuf directory. 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 page of the Component 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:

```
Videoplayer1.Play;
```

- 10 Choose Run | Run to execute the AVI video.

Writing multi-threaded applications

The VCL provides several objects that 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 95 only simulates multiprocessing, even if the underlying hardware supports it.

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” on page 8-6. For more information on using *BeginThread*, see the online help.

To use a thread object in your application, you must create a new descendant of *TThread*. To create a descendant of *TThread*, choose File | New from the main menu. In the new objects dialog box, select Thread Object. You are prompted to provide a class name for your new thread object. After you provide the name, Delphi creates a new unit file to implement the thread.

Note Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog does not automatically prepend a ‘T’ to the front of the class name you provide.

The automatically generated file contains the skeleton code for your new thread object. If you named your thread *TMyThread*, it would look like the following:

```
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.
```

You must fill in the code for the *Execute* method. These steps are described in the following sections.

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. *Priority* values fall along a seven point scale, as described in Table 8.1:

Table 8.1 Thread priorities

Value	Priority
<code>tpIdle</code>	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with <i>tpIdle</i> priority.
<code>tpLowest</code>	The thread's priority is two points below normal.
<code>tpLower</code>	The thread's priority is one point below normal.
<code>tpNormal</code>	The thread has normal priority.
<code>tpHigher</code>	The thread's priority is one point above normal.
<code>tpHighest</code>	The thread's priority is two points above normal.
<code>tpTimeCritical</code>	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:

```

constructor TMyThread.Create(CreateSuspended: Boolean);
{
    inherited Create(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.

Using the main VCL thread

When you use objects from the VCL object 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 VCL thread is set aside for access of 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 VCL thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

```

procedure TMyThread.PushTheButton;
begin
    Button1.Click;
end;
:
procedure TMyThread.Execute;
begin
    :
    Synchronize(PushTheButton);
    :
end;

```

Synchronize waits for the main VCL 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 VCL 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 in the following situations:

- Data access components are thread-safe as long as each thread has its own database session component. The one exception to this is when you are using Access drivers. Access drivers are built using the Microsoft ADO library, which is not 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, see "Managing multiple sessions" on page 16-16.

- Graphics objects are thread-safe. You do not need to use the main VCL thread to access *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile*, or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them (see “Locking objects” on page 8-6).
- While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

Using thread-local variables

Your *Execute* method and any of the routines it calls have their own local variables, just like any other Object Pascal 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. For example,

```
threadvar
  x : integer;
```

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.

Checking for termination by other threads

Your thread begins running when the *Execute* method is called (see “Executing thread objects” on page 8-10) 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:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

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 components and VCL objects from the *OnTerminate* event handler without worrying about clashing with other threads.

For more information about the main VCL thread, see “Using the main VCL thread” on page 8-4.

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.

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.

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.

The VCL also includes a thread-safe list object, *TThreadList*. Calling *TThreadList.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:

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

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 one which 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 objects in the VCL, use the main VCL thread to execute your code. Using the main VCL 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” on page 8-4 for more information on the main VCL 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” on page 8-5 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 you can wait for another thread to signal that it has completed a task.

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:

```

if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
    end;
  ThreadList1.UnlockList;
end;

```

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.

```
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;
```

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 Table 8.2.

Table 8.2 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.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
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 }
```

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:

```
SecondProcess := TMyThread.Create(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.

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" on page 8-2. 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:

```
SecondProcess := TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.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*.

Using threads in distributed applications

Distributed applications introduce additional challenges for writing multi-threaded applications. When considering how to coordinate threads, you must also keep in mind how other processes affect the threads in your application.

Usually, handling distributed threading issues is the responsibility of the server application. When writing servers, you must consider how requests from clients are serviced.

If each client request has its own thread, you must ensure that different client threads do not interfere with each other. In addition to the usual issues that arise when coordinating multiple threads, you may need to ensure that each client has a consistent view of your application. For example, you can't use thread variables to store information that must persist over multiple client requests if each time the client calls your application it uses a different thread. When clients change the values of object properties or global variables, they are influencing not only their own view of that object or variable, but the view of any other clients.

Using threads in message-based servers

Message-based servers receive client request messages, perform some action in response to that message, and return messages to the client. Examples include internet server applications and simple services that you can write using sockets.

Usually, when writing message-based servers, each client message gets its own thread. When client messages are received, the application spawns a thread to handle the message. This thread runs until it sends a response to the client, and then terminates. You must be careful when using global objects and variables, but it is fairly easy to control how threads are created and run because client messages are all received and dispatched by the main application thread.

Using threads with distributed objects

When writing servers for distributed objects, the threading issues are more complicated. Unlike message-based servers, where there is a point in the code where messages are received and dispatched, clients call into server objects by calling any of their methods or by accessing any of their properties. Because of this, there is no easy way for server applications to spawn separate threads for each client request.

Writing applications (.EXEs)

When writing an .EXE that implements an object or objects for remote clients, client requests come in as threads. How this works depends on whether clients access your object using COM or CORBA.

- **Under COM**, client requests come in as part of the application's message loop. This means that any code which executes after the application's main message loop starts up must be prepared to protect access to objects and global memory

from other threads. When running in an environment that supports DCOM, Delphi ensures that no client requests occur until all code in the initialization part of your units has executed. If you are not running in an environment that supports DCOM, you must ensure that any code in the initialization part of your units is thread-safe.

- **Under CORBA**, you can choose a threading model in the wizard that starts a new CORBA server. You can choose either single-threading or multi-threading. Under both models, each client connection has its own thread. You can use thread variables for information that persists across client calls because all calls for a given client use the same thread. With single-threading, only one client thread has access to an object instance at a time. While you must protect access to global memory, you are assured of no conflicts when accessing the object's instance data (such as property values). With multi-threading, multiple clients may access your application at the same time. If you are sharing object instances over clients, you must protect both global data and instance data.

Writing libraries

When an Active Library implements the distributed object, threading is usually controlled by the technology (COM, DCOM, or MTS) that supports distributed object calls. When you first create your server library with the appropriate wizard, you are prompted to specify a threading model that dictates how client requests are assigned threads. These models include the following:

- **Single-threaded model.** Client requests are serialized by the calling mechanism. Your .DLL does not need to be concerned with threading issues because it receives one client request at a time.
- **Single-threaded apartment model.** (Also called Apartment model.) Each object instantiated by a client is accessed by one thread at a time. You must protect against multiple threads accessing global memory, but instance data (such as object properties) is thread-safe. Further, each client always accesses the object instance using the same thread, so that you can use thread variables.
- **Activity model.** (Called both Apartment-threaded and Free-threaded under MTS.) Each object instance is accessed by one thread at a time, but clients do not always use the same thread for every call. Instance data is safe, but you must guard global memory, and thread variables will not be consistent across client calls.
- **Multi-threaded apartment model.** (Also called Free-threading.) Each object instance may be called by multiple threads simultaneously. You must protect instance data as well as global memory. Thread variables are not consistent across client calls.
- **Single/Multi-threaded apartment model.** (Also called Both.) This is the same as the Multi-threaded apartment model, except that all callbacks supplied by clients are guaranteed to execute in the same thread. This means you do not need protect values supplied as parameters to callback functions.

Note Typically, a wizard assigns a threading model to your object. When you add multiple COM objects to an EXE, the application initializes COM with the highest level of thread support indicated (where single-threaded is the lowest and Both is highest).

You can manually override the way your application initializes COM threading support by changing the global *CoInitFlags* variable in the program's main source file before the call to *Application.Initialize*.

COM-based systems use the application's message loop to synchronize threads in all but the Multi-threaded apartment model (which is only available under DCOM). Because of this, you must ensure that any lengthy call made through a COM interface calls the application object's *ProcessMessages* method. Failure to do so prevents other clients from gaining access to your application, effectively making your library single-threaded.

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 | 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*.

For additional details on the Thread Status box, see online Help.

Working with packages and components

A *package* is a special dynamic-link library used by Delphi 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 Delphi components reside in a package called VCL50. Each time you create an application, you can specify that it uses VCL50. 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 VCL50.BPL. A computer with several package-enabled applications installed on it needs only a single copy of VCL50.BPL, which is shared by all the applications and the IDE itself.

Delphi ships with several precompiled runtime packages, including VCL50, that encapsulate VCL components. Delphi also uses design-time packages to 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 Windows application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages

Table 9.1 Compiled package files

File extension	Contents
.DPK	The source file listing the units contained in the package.
DCP	A binary image containing a package header and the concatenation of all DCU files in the package, including all symbol information required by the compiler. A single DCP file is created for each package. The base name for the DCP is the base name of the DPK source file. You must have a .DCP file to build an application with packages.
DCU	A binary image for a unit file contained in a package. One DCU is created, when necessary, for each unit file.
BPL	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the BPL is the base name of the DPK source file.

Note Packages share their global data with other modules in an application.

For more information about DLLs and packages, see the *Object Pascal Language Guide*.

Runtime packages

Runtime packages are deployed with Delphi 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 .EXE 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.

Using packages in an application

To use packages in an application,

- 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. (Runtime packages associated with installed design-time packages are already listed in the edit box.) 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 will be changing Delphi’s global Library Path.

You do not need to include file extensions with package names. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons. For example:

```
VCL50;VCLDB50;VCLDBX50
```

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 edit box is empty 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 still need to 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:

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

Each of the units referenced in this example is contained in the VCL50 package. Nonetheless, you must keep these references in the **uses** clause, even if you use VCL50 in your application, or you will get compiler errors. In generated source files, Delphi adds these units to the **uses** clause automatically.

Dynamically loading packages

To load a package at runtime, call the *LoadPackage* function. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```

with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(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

Delphi ships with several precompiled runtime packages, including VCL50, which supply basic language and component support.

The VCL50 package contains the most commonly used components, system functions, and Windows interface elements. It does not include database or Windows 3.1 components, which are available in separate packages. The following table lists some runtime packages shipped with Delphi and the units they contain.

Table 9.2 Runtime packages

Package	Units
VCL50.BPL	Ax, Buttons, Classes, Clipbrd, Comctrls, Commctrl, Commdlg, Comobj, Comstrs, Consts, Controls, Ddeml, Dialogs, Dlgs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Forms, Graphics, Grids, Imm, IniFiles, Isapi, Isapi2, Istreams, Libhelp, Libintf, Lzexpand, Mapi, Mask, Math, Menu, Messages, Mmsystem, Nsapi, Ole2L, Oleconst, Olectnrs, Olectrls, Oledlg, Penwin, Printers, Proxies, Registry, Regstr, Richedit, Shellapi, Shlobj, Stdctrls, Stdvcl, Sysutils, Tlhelp32, Toolintf, Toolwin, Typinfo, Vclcom, Virtintf, Windows, Wininet, Winsock, Winspool, Winsvc
VCLX50.BPL	Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs
VCLDB50.BPL	Bde, Bdeconst, Bdeprov, Db, Dbcgrids, Dbclient, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables, Dsintf, Provider, SMintf
VCLDBX50.BPL	Dblookup, Report
DSS50.BPL	Mxarrays, Mxbutton, Mxcommon, Mxconsts, Mxdb, Mxdcube, Mxdssqry, Mxgraph, Mxgrid, Mxpivsrc, Mxqedcom, Mxqparse, Mxqryedt, Mxstore, Mxtables, Mxqvb
QRPT50.BPL	Qr2const, Qrabout, Qralias, Qrctrls, Qrdatasu, Qrexpbld, Qrextra, Qrprev, Qrprgres, Qrprntr, Qrqred32, Quickrpt
TEE50.BPL	Arrowcha, Bubblech, Chart, Ganttch, Series, Teeconst, Teefunci, Teengine, Teeprocs, Teeshape
TEEDB50.BPL	Dbchart, Qrtee
TEEU50.BPL	Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Brushdlg, Bubbledi, Custedit, Dbeditch, Editchar, Flineedi, Ganttedi, Ieditcha, Pendlg, Pieedit, Shapeedi, Teeabout, Teegally, Teelish, Teeprevi, Teexport
VCLSMP50.BPL	Sampreg, Smpconst

To create a client/server database application that uses packages, you need at least two runtime packages: VCL50 and VCLDB50. If you want to use Outline components

in your application, you also need VCLX50. To use these packages, choose Project | Options, select the Packages tab, and enter the following list in the Runtime Packages edit box.

```
VCL50;VCLDB50;VCLX50
```

Actually, you don't have to include VCL50, because VCL50 is referenced in the Requires clause of VCLDB50. (See "The Requires clause" on page 9-9.) Your application will compile just the same whether or not VCL50 is included in the Runtime Packages edit box.

Custom packages

A custom package is either a BPL you code and compile yourself, or a precompiled 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:

```
VCL50;VCLDB50;STATS
```

If you create your own packages, you can add them to the list as needed.

Design-time packages

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Delphi ships with the following design-time component packages preinstalled in the IDE.

Table 9.3 Design-time packages

Package	Component palette pages
DCLSTD50.BPL	Standard, Additional, System, Win32, Dialogs
DCLTEE50.BPL	Additional (<i>TChart</i> component)
DCLDB50.BPL	Data Access, Data Controls
DCLMID50.BPL	Data Access (MIDAS)
DCL31W50.BPL	Win 3.1
DCLNET50.BPL	Internet
NMFAST50.BPL	
DCLSMP50.BPL	Samples
DCLOCX50.BPL	ActiveX
DCLQRT50.BPL	QReport
DCLDSS50.BPL	Decision Cube
IBSMP50.BPL	Samples (<i>IBEventAlerter</i> component)
DCLINT50.BPL	(International Tools—Resource DLL wizard)

Table 9.3 Design-time packages (continued)

Package	Component palette pages
RCEXPRT.BPL	(Resource Expert)
DBWEBXPRT.BPL	(Web Wizard)

These design-time packages work by calling runtime packages, which they reference in their Requires clauses. (See “The Requires clause” on page 9-9.) For example, DCLSTD50 references VCL50. DCLSTD50 itself contains additional functionality that makes most of the standard components available on the Component palette.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The DCLUSR50 design-time package is provided as a default container for new components.

Installing component packages

In Delphi all components are installed in the IDE as packages. If you’ve written your own components, create and compile a package that contains them. (See “Creating and editing packages” on page 9-7.) Your component source code must follow the model described in Part IV, “Creating custom components”.

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

- 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. (For information about these files, see “Package files created by a successful compilation” on page 9-13.)

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 need be copied; the .DPC file contains the other files. (For more information about package collection files, see “Package collection files” on page 9-13.)

- 2 Choose Component | Install Packages from the IDE menu, or choose Project | Options and click the Packages tab.
- 3 A list of available packages appears under “Design packages”.
 - To install a package in the IDE, select the check box next to it.
 - To uninstall a package, deselect 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 Open Package dialog box for the directory where the .BPL or .DPC 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.

4 Click OK.

The components in the package are installed on the Component 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 default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the dialog box.

To remove components from the Component palette without uninstalling a package, select Component | Configure Palette, or select Tools | Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

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, which contain the functionality of the compiled .BPL. The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Package source files, which end with the .DPK extension, are generated by the Package editor.

Creating a package

To create a package, follow the procedure below. Refer to “Understanding the structure of a package” on page 9-9 for more information about the steps outlined here.

- 1 Choose File | New, select the Package icon, and click OK.
- 2 The generated package is displayed in the Package editor.
- 3 The Package editor shows a *Requires* node and a *Contains* node for the new package.
- 4 To add a unit to the **contains** clause, click the Add to package speed button. 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 Package editor. You can add additional units by repeating this step.

- 5 To add a package to the **requires** clause, click the Add to package speed button. 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 Package editor. You can add additional packages by repeating this step.
- 6 Click the Options speed button, and decide what kind of package you want to build.
 - To create a design-time-only package (a package that cannot be used at runtime), select the Designtime only radio button. (Or add the `{$DESIGNONLY}` compiler directive to the DPK file.)
 - To create a runtime-only package (a package that cannot be installed), select the Runtime only radio button. (Or add the `{$RUNONLY}` compiler directive to the DPK file.)
 - To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.
- 7 In the Package editor, click the Compile package speed button to compile your package.

Editing an existing package

There are several ways to open an existing package for editing.

- Choose File | Open (or File | Reopen) and select a DPK file.
- Choose Component | Install Packages, select a package from the Design Packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options speed button in the Package editor and select the Description tab.

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.

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 Object Pascal source code.

To open a package source file in the Code editor,

- 1 Open the package in the Package editor.

2 Right-click in the Package editor and select 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 code declares the VCLDB50 package.

```
package VCLDB50;
  requires VCL50;
  contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreg, Dblogdlg, Dbpwdlg, Dbtables,
mycomponent in 'C:\components\mycomponent.pas';
end.
```

Understanding the structure of a package

Naming packages

Package names must be unique within a project. If you name a package **STATS**, the Package editor 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.

The 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 will require **VCL50**. Any package that depends on **VCL** units (including **SysUtils**) must list **VCL50**, or another package that requires **VCL50**, in its **requires** clause.

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 in the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

The 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 **contains** clause 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 Delphi IDE*. This means that if you create a package that contains one of the units in VCL50, 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.

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.
- 2 Select Delphi Package Source (*.DPK) from the Files Of Type drop-down list.
- 3 Select a .DPK file in the dialog.
- 4 When the Package editor opens, click the Compile speed button.

You can insert compiler directives into your package source code. For more information, see “Package-specific compiler directives”, below.

If you compile from the command line, several package-specific switches are available. For more information, see “Using the command-line compiler and linker” on page 9-12.

Package-specific compiler directives

The following table lists package-specific compiler directives that you can insert into your source code.

Table 9.4 Package-specific compiler directives

Directive	Purpose
{SIMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .DPK files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{G-} or {IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{WEAKPACKAGEUNIT ON}	Packages unit “weakly.” See “Weak packaging” on page 9-11 below.
{DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .DPK file.)
{RUNONLY ON}	Compiles the package as runtime only. (Put in .DPK 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 in the online help for information on compiler directives not discussed here.

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 by a successful compilation” on page 9-13.) 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 PACK that contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.DLL. If you put **{WEAKPACKAGEUNIT ON}** in UNIT1.PAS when you compile your package, UNIT1 will not be included in PACK.BPL; you will not have to distribute copies of RARE.DLL with PACK. However, UNIT1 will still be included in PACK.DCP. If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.DCP and compiled directly into the project.

Now suppose you add a second unit, UNIT2, to PACK. Suppose that UNIT2 uses UNIT1. This time, even if you compile PACK with **{WEAKPACKAGEUNIT ON}** in

UNIT1.PAS, the compiler will include UNIT1 in PACK.BPL. But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.DCP.

Note Unit files containing the `{SWEAKPACKAGEUNIT ON}` directive must not have global variables, initialization sections, or finalization sections.

The `$SWEAKPACKAGEUNIT` directive is an advanced feature intended for developers who distribute their BPLs to other Delphi 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, Delphi's 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 VCL50. When you compile a project that uses PenWin and the VCL50 package, PenWin is copied from VCL50.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, VCL50 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 VCL50 and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of VCL50.

Using the command-line compiler and linker

When you compile from the command line, you can use the package-specific switches listed in the following table.

Table 9.5 Package-specific command-line compiler switches

Switch	Purpose
<code>-\$G-</code>	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.
<code>-LEpath</code>	Specifies the directory where the package BPL file will be placed.
<code>-LNpath</code>	Specifies the directory where the package DCP file will be placed.
<code>-LUpackage</code>	Use packages.
<code>-Z</code>	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" in the online help for information on command-line options not discussed here.

Package files created by a successful compilation

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.

The following table lists the files produced by the successful compilation of a package.

Table 9.6 Compiled package files

File extension	Contents
DCP	A binary image containing a package header and the concatenation of all DCU files in the package. A single DCP file is created for each package. The base name for the DCP is the base name of the DPK source file.
DCU	A binary image for a unit file contained in a package. One DCU is created, when necessary, for each unit file.
BPL	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the BPL is the base name of the DPK source file.

Deploying packages

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 a package collection,

- 1 Choose Tools | Package Collection Editor to open the Package Collection editor.
- 2 Click the Add Package speed button, then select a BPL in the Select Package dialog and click Open. To add more BPLs to the collection, click the Add Package speed 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 click the Remove Package speed button.
- 3 Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields will appear:
 - In the Author/Vendor Name edit box, you can enter optional information about your package collection that will 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 File Group speed 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 Delphi's 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 Delphi's 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 speed 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.

Creating international applications

This chapter 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.

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

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 a large set of locales, each of which is described by a language and country pair.

Localization

Localization is the process of translating an application to function in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified to be aware of the different tax laws in different countries.

Internationalizing applications

It is not difficult to create internationalized applications. You need to complete the following steps:

- 1 You must enable your code to handle strings from international character sets.
- 2 You need to design your user interface so that it can accommodate the changes that result from localization.
- 3 You need to 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.

Character sets

The United States edition of Windows uses 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 1- or 2-byte character codes.

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).

Double byte 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 1-byte *char*. Instead, characters are represented by a mix of 1- and 2-byte character codes.

The first byte of every 2-byte character code is taken from a reserved range that depends on the specific character set. The second byte can sometimes be the same as the character code for a separate 1-byte character, or it can fall in the range reserved for the first byte of 2-byte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or part of a 2-byte character is to read the string, starting at the beginning, parsing it into 2-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 1- and 2-byte

characters. Delphi provides you with a number of runtime library functions that allow you to do this. These functions are as follows:

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 2-byte 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

Another approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Wide characters are two bytes instead of one, so that 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 95 only 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 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*.

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. The following table lists VCL objects that have these properties:

Table 10.1 VCL objects that support BiDi

Component palette page	VCL object
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
TScrollBar	
Additional	TBitBtn
	TCheckListBox
	TDrawGrid
	TMaskEdit
	TScrollBar
	TSpeedButton
	TStaticLabel
TStringGrid	
Win32	TDateTimePicker
	THeaderControl
	TListView
	TMonthCalendar
	TPageControl
	TStatusBar

Table 10.1 VCL objects that support BiDi (continued)

Component palette page	VCL object
	TTabControl
Data Controls	TDBCheckBox TDBComboBox TDBEdit TDBGrid TDBListBox TDBLookupComboBox TDBLookupListBox TDBMemo TDBRadioGroup TDBRichEdit TDBText
QReport	TQRDBText TQRExpr TQRLabel TQRMemo TQRSysData
Other classes	TApplication (has no <i>ParentBiDiMode</i>) TForm THintWindow (has no <i>ParentBiDiMode</i>) TStatusPanel THeaderSection

Notes *THintWindow* picks up the *BiDiMode* of the control that activated the hint.

Bi-directional properties

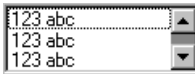
The objects listed in Table 10.1, “VCL objects that support BiDi,” on page 10-4 have the properties *BiDiMode* and *ParentBiDiMode*. These properties, along with *TApplication’s BiDiKeyboard* and *NonBiDiKeyboard*, support bi-directional localization.

BiDiMode property

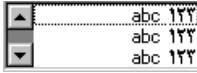
The property *BiDiMode* is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly*.

bdLeftToRight

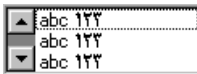
bdLeftToRight draws text using left to right reading order, and the alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

Figure 10.1 TListBox set to `bdLeftToRight`**bdRightToLeft**

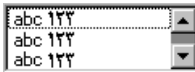
bdRightToLeft draws text using right to left reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left-to-right.

Figure 10.2 TListBox set to `bdRightToLeft`**bdRightToLeftNoAlign**

bdRightToLeftNoAlign draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

Figure 10.3 TListBox set to `bdRightToLeftNoAlign`**bdRightToLeftReadingOnly**

bdRightToLeftReadingOnly draws text using right to left reading order, and the alignment and scroll bars are not changed.

Figure 10.4 TListBox set to `bdRightToLeftReadingOnly`**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.

Method	Description
<code>OkToChangeFieldAlignment</code>	Used with database controls. Checks to see if the alignment of a control can be changed.
<code>DBUseRightToLeftAlignment</code>	A wrapper for database controls for checking alignment.
<code>ChangeBiDiModeAlignment</code>	Changes the alignment parameter passed to it. No check is done for <i>BiDiMode</i> setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
<code>IsRightToLeft</code>	Returns <i>True</i> if any of the right to left options are selected. If it returns <i>False</i> the control is in left to right mode.
<code>UseRightToLeftReading</code>	Returns <i>True</i> if the control is using right to left reading.
<code>UseRightToLeftAlignment</code>	Returns <i>True</i> if the control is using right to left alignment. It can be overridden for customization.
<code>UseRightToLeftScrollBar</code>	Returns <i>True</i> if the control is using a left scroll bar.
<code>DrawTextBiDiModeFlags</code>	Returns the correct draw text flags for the <i>BiDiMode</i> of the control.
<code>DrawTextBiDiModeFlagsReadingOnly</code>	Returns the correct draw text flags for the <i>BiDiMode</i> of the control, limiting the flag to its reading order.
<code>AddBiDiModeExStyle</code>	Adds the appropriate <i>ExStyle</i> 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.

VCL components offer you 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. *TWinControl* 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.

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.

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. Table 10.2 provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Table 10.2 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, 2-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 .DFM file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the *Object Pascal Language Guide*. It is best to include all resource strings in a single, separate unit.

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 obtain the locale code for the target translation:

```

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);
begin
  EnumSystemLocales(@LocalesCallback, LCID_SUPPORTED);
end;

end.

```


Using resource DLLs

The executable, DLLs, and packages 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 EXE, DLL, or BPL 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 Delphi applications:

```

procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocaleOverride, FileName);
  finally
    Reg.Free;
  end;

```

Within your application, use the global `FindResourceHInstance` function to obtain the handle of the current resource module. For example:

```

LoadStr(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:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  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 reacquire 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 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 DFM files and a RES 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.

If your version of Delphi includes the Integrated Translation Environment, you can use the ITE to manage localization. For more information, see the online Help for the ITE.

Deploying applications

Once your Delphi 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 those steps for deploying 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. These are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files

Delphi applications that access databases and those that run across the Web require additional installation steps beyond those that apply to general applications. For additional information on installing database applications, see "Deploying database applications" on page 11-4. For more information on installing Web applications, see "Deploying Web applications" on page 11-7. For more information on installing ActiveX controls, see "Deploying an ActiveX control on the Web" on page 48-17. For

information on deploying CORBA applications, see “Deploying CORBA applications” on page 28-16.

Using installation programs

Simple Delphi 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 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. InstallShield Express is not automatically installed when Delphi is installed, and must be manually installed to be used 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, you should only use those 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
- ActiveX controls

Application files, listed by file name extension

The following types of files may need to be distributed with an application.

Table 11.1 Application files

Type	File name extension
Program files	.EXE and .DLL
Package files	.BPL and .DCP
Help files	.HLP, .CNT, and .TOC (if used)
ActiveX files	.OCX (sometimes supported by a DLL)
Local table files	.DBF, .MDX, .DBT, .NDX, .DB, .PX, .Y*, .X*, .MB, .VAL, .QBE

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. 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 both the .BPL and the .DCP files.

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 of your own creation 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, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows 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 Delphi application would be partially or completely unable to function. Helper applications may be those supplied with Windows, by Borland, or they might be 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 documentation for the helper 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 even more specialized handling on installation, because the files that make up the application are typically located on multiple computers. Two ways of including database access are

- Providing the database engine
- Multi-tiered Distributed Application Services (MIDAS)

Providing the database engine

Database access for an application is provided by various database engines. An application can use the Borland Database Engine or a third-party database engine. SQL Links is provided (not available in all versions) to enable native access to SQL database systems. The following sections describe installation of the database access elements of an application:

- Borland Database Engine
- Third-party database engines
- SQL Links

Borland Database Engine

For standard Delphi data components to have database access, the Borland Database Engine (BDE) must be present and accessible. See BDEDEPLOY.TXT for specific rights and limitations on redistributing the BDE.

Borland recommends use of InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express will create the necessary registry entries and define 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 Windows 95 and Windows NT, 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.

Third-party database engines

You can use third-party database engines to provide database access for Delphi applications. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

SQL Links

SQL Links provides the drivers that connect an application (through the Borland Database Engine) with the client software for an SQL database. See DEPLOY.TXT for specific rights and limitations on redistributing SQL Links. As is the case with the Borland Database Engine (BDE), SQL Links must be deployed using InstallShield Express (or other certified installation program).

Note SQL Links only connects the BDE to the client software, not to the SQL database itself. It is still necessary to install the client software for the SQL database system used. See the documentation for the SQL database system or consult the vendor that supplies it for more information on installing and configuring client software.

Table 11.2 shows the names of the driver and configuration files SQL Links uses to connect to the different SQL database systems. These files come with SQL Links and are redistributable in accordance with the Delphi license agreement.

Table 11.2 SQL database client software files

Vendor	Redistributable files
Oracle 7	SQLORA32.DLL and SQL_ORA.CNF
Oracle8	SQLORA8.DLL and SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL and SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL and SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL and SQL_MSS.CNF
Informix 7	SQLINF32.DLL and SQL_INF.CNF
Informix 9	SQLINF9.DLL and SQL_INF9.CNF
DB/2	SQLDB232.DLL and SQL_DB2.CNF
InterBase	SQLINT32.DLL and SQL_INT.CNF

Install SQL Links using InstallShield Express or other certified installation program. For specific information concerning the installation and configuration of SQL Links, see the help file SQLLNK32.HLP, by default installed into the main BDE directory.

Multi-tiered Distributed Application Services (MIDAS)

Multi-tiered Distributed Application Services (MIDAS) consists of the Business Object Broker, OLEnterprise, the Remote DataBroker, and the ConstraintBroker

Manager (SQL Explorer). MIDAS provides multi-tier database capability to Delphi applications.

Handle the installation of the executable and related files for a multi-tier application the same as for general applications. Some of the files that comprise MIDAS may need to be installed on the client computer and others on the server computer. For general application installation information, see See “Deploying general applications” on page 11-1. See the text file LICENSE.TXT on the MIDAS CD and the Delphi file DEPLOY.TXT for specific information regarding licensing and redistribution rights for MIDAS.

MIDAS.DLL must be installed onto the client computer and registered with Windows. On the server computer, the files MIDAS.DLL and STDVCL40.DLL must be installed and registered for the Remote DataBroker and DBEXPLOR.EXE for the ConstraintBroker. Installation programs such as InstallShield Express automate the process of registering these DLLs. To manually register the DLLs, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

The MIDAS deployment CD provides install programs for the client and server portions of OLEnterprise and the Business ObjectBroker. Use only the Setup Launcher on the MIDAS CD to install OLEnterprise.

Following is a list of the minimum required files to be installed onto the server machine.

UNINSTALL.EXE	OBJFACT.ICO	W32PTHD.DLL	NBASE.IDL
LICENSE.TXT	ODEBKN40.DLL	RPMARN40.DLL	OBJX.EXE
README.TXT	ODECTN40.DLL	RPMAWN40.DLL	OLECFG.EXE
OLENTER.HLP	RPMEGN40.DLL	RPMCBN40.DLL	OLEWAN40.CAB
OLENTER.CNT	ODEDIN40.DLL	RPMCPCN40.DLL	OLENTEXP.EXE
FILELIST.TXT	ODEEGN40.DLL	BROKER.EXE	OLENTEXP.HLP
SETLOG.TXT	ODELTN40.DLL	RPMFEN40.DLL	OLENTEXP.CNT
SETLOG.EXE	LIBAVEMI.DLL	RPMUTN40.DLL	BRKCP.EXE
OBJPING.EXE	OLEAAN40.DLL	RPMFE.CAT	BROKER.ICO
OBJFACT.EXE	OLERAN40.DLL	EXPERR.CAT	

Following is a list of the required files to be installed onto the client machine.

NBASE.IDL	ODEN40.DLL	RPMFEN40.DLL	OLENTEXP.EXE
ODECTN40.DLL	RPMARN40.DLL	RPMUTN40.DLL	SETLOG.EXE
ODEDIN40.DLL	RPMAWN40.DLL	OLERAN40.DLL	OLECFG.EXE
ODEEGN40.DLL	RPMCBN40.DLL	OLEAAN40.DLL	W32PTHD.DLL
ODELTN40.DLL	RPMCPCN40.DLL	OLEWAN40.CAB	
ODEMSG.DLL	RPMEGN40.DLL	OBJX.EXE	

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 (ISAPI) DLLs, CGI applications, and ActiveForms.

The steps for installing Web applications are the same as those for general applications, except the application's files are deployed on the Web server. For information on installing general applications, see See "Deploying general applications" on page 11-1.

Here are some special considerations for deploying Web applications:

- For database applications, the Borland Database Engine (or alternate database engine) is installed along with the application files on the Web server.
- Security for the directories must not be so high that access to application files, the BDE, or database files is not possible.
- 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.

Programming for varying host environments

Due to the nature of the Windows environment, 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
- Windows versions
- Helper applications
- DLL locations

Screen resolutions and color depths

The size of the Windows 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.

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 Windows 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 is 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 the *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* property to *True* and calling the *TWincontrol.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 Windows displays it 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. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard Windows set.

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 an application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .FOT file for the nonstandard font with the application.

Windows versions

When using Windows API functions or accessing areas of the Windows operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different versions of Windows. For example, Services are only pertinent to the Windows NT operating system. If an application is to act as a Service or interact with one, this would fail if the application is installed under Windows 95.

To account for this possibility, you have a few options:

- Specify in the application's system requirements the versions of Windows on which the application can run. It is the user's responsibility to install and use the application only under compatible Windows versions.
- Check the version of Windows as the application is installed. If an incompatible version of Windows is present, either halt the installation process or at least warn the installer of the problem.
- Check the Windows version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of Windows is present, abort the process and alert the user. Alternately, provide different code to run dependent on different versions of Windows. Some operations are performed differently in Windows 95 than in Windows NT. 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 where limitations exist:

- DEPLOY.TXT
- README.TXT
- No-nonsense license agreement
- Third-party product documentation

DEPLOY.TXT

DEPLOY.TXT 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 C++BuilderDelphi application. DEPLOY.TXT is a text file installed in the main C++BuilderDelphi directory. The topics covered include

- .EXE, .DLL, and .BPL files
- Components and design-time packages
- Borland Database Engine (BDE)
- ActiveX controls
- Sample Images
- Multi-tiered Distributed Application Services (MIDAS)
- SQL Links

README.TXT

README.TXT contains last minute information about C++BuilderDelphi, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README.TXT is a Windows help file installed into the main C++BuilderDelphi 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

The chapters in “Developing Database Applications” present concepts and skills necessary for creating Delphi database applications.

Note You need the Professional or Enterprise edition of Delphi to develop database applications. To implement more advanced Client/Server databases, you need the Delphi features available in the Enterprise edition.

Designing database applications

Database applications allow users to 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.

This chapter introduces some common considerations for designing a database application and the decisions involved in designing a user interface.

Using databases

The components on the Data Access page, the ADO page, or the InterBase page of the Component palette allow your application to read from and write to databases. The components on the Data Access page use the Borland Database Engine (BDE) to access database information which they make available to the data-aware controls in your user interface. The components on the ADO page use ActiveX Data Objects (ADO) to access the database information through OLEDB. The components on the InterBase page access an InterBase database directly.

Depending on your version of Delphi, the BDE includes drivers for different types of databases. While all types of databases contain tables which store information, different types support additional features such as

- Database security
- Transactions
- Data dictionary
- Referential integrity, stored procedures, and triggers

Types of databases

You can connect to different types of databases, depending on what drivers you have installed with the BDE or ADO.

These drivers may connect your application to local databases such as Paradox, Access, and dBASE or remote database servers like Microsoft SQL Server, Oracle, and Informix. Similarly, the InterBase Express components can access either a local or remote version of InterBase.

Note Different versions of Delphi come with the components that use these drivers (BDE or ADO), or with the InterBase express components.

Choosing what type of database to use depends on several factors. Your data may already be stored in an existing database. If you are creating the tables of information your application uses, you may want to consider the following questions.

- How much data will the tables hold?
- How many users will be sharing these tables?
- What type of performance (speed) do you require from the database?

Local databases

Local databases reside on your local drive or on a local area network. They have proprietary APIs for accessing the data. Often, they are dedicated to a single system. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases.

Local databases can be faster than remote database servers because they often reside on the same system as the database application.

Because they are file-based, local databases are more limited than remote database servers in the amount of data they can store. Therefore, in deciding whether to use a local database, you must consider how much data the tables are expected to hold.

Applications that use local databases are called single-tiered applications because the application and the database share a single file system.

Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Remote database servers

Remote database servers usually reside on a remote machine. They use Structured Query Language (SQL) to enable clients to access the data. Because of this, 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.

Remote database servers are designed for access by several users at the same time. Instead of a file-based locking system such as those employed by local databases, they provide more sophisticated multi-user support, based on transactions.

Remote database servers hold more data than local databases. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers.

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).

Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

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 when using the BDE, see “Controlling server login” on page 17-6. For information on providing this information when using ADO, see “Controlling the connection login” on page 23-7. For information on providing this information when using the InterBase direct access components, see the *OnLogin* event of *TIBDatabase*.

When designing database applications, you must consider what type of authentication is required by your database server. If you do not want your users to have to provide a 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 are requiring 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 before you access the table, even though it is not required until then.

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 which 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, CORBA, or MTS 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 protect against hardware failures that occur in the middle of a database command or set of commands. They 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.

Although transaction support is not part of most local databases, the BDE drivers provide limited transaction support for some of these databases. For SQL servers and ODBC-compliant databases, the database transaction support is provided by the component that represents the database connection. 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 BDE-based database applications, see "Using transactions" on page 13-5. For details on using transactions in ADO-based database applications, see "Working with (connection) transactions" on page 23-11. For details on using transactions in multi-tiered applications, see "Managing transactions in multi-tiered applications" on page 14-25. For details on using transactions in applications that use the InterBase direct access components, see the online help for the *TIBTransaction* component.

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" on page 19-14. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the `drntf` unit (located in the `lib` directory). This interface supplies the following methods:

Table 12.1 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.

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 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 manage the database or databases, and components that represent the data contained by the tables in those databases (datasets). How you organize these pieces is the architecture of your database application.

By isolating database access components in data modules, you can develop forms in your database applications that provide a consistent user interface. By storing links to well-designed forms and data modules 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 and modules also makes it possible for you to develop corporate standards for database access and application interfaces.

Many aspects of the architecture of your database application depend on the type of database you are using, the number of users who will be sharing the database information, and the type of information you are working with. See “Types of databases” on page 12-2 for more information about different types of databases.

When writing applications that use information that is not shared among several users, you may want to use a local database in a *single-tiered application*. This approach can have the advantage of speed (because data is stored locally), and does not require the purchase of a separate database server and expensive site licences. However, it is limited in how much data the tables can hold and the number of users your application can support.

Writing a *two-tiered application* provides more multi-user support and lets you use large remote databases that can store far more information.

Note Support for two-tiered applications requires SQL Links, InterBase, or ADO.

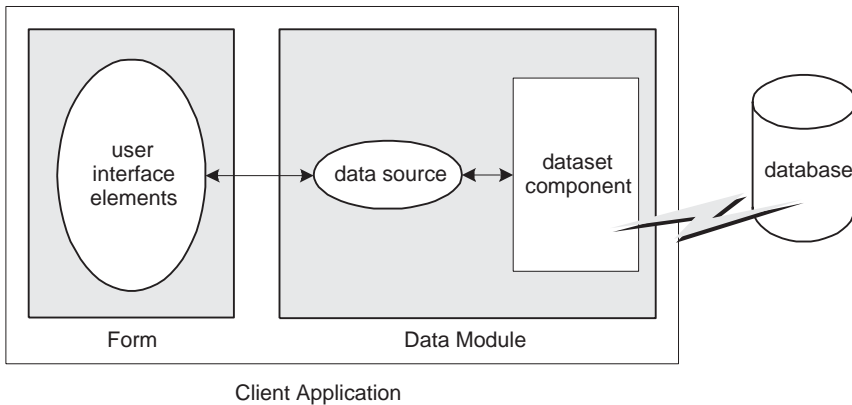
When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a *multi-tiered application*. Multi-tiered applications include middle tiers that centralize the logic which governs your database interactions so that there is centralized control over data relationships. This allows different client applications to use the same data while ensuring that the data logic is consistent. They also allow for smaller client applications because much of the processing is off-loaded onto middle tiers. These smaller client applications are easier to install, configure, and maintain because they do not include the database connectivity software. Multi-tiered applications can also improve performance by spreading the data-processing tasks over several systems.

Planning for scalability

The development process can get more involved and expensive as the number of tiers increases. Because of this, you may wish to start developing your application as a single-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. By planning for scalability, you can protect your development investment when writing a single- or two-tiered application so that the code can be reused as your application grows.

The VCL data-aware components make it easy to write scalable applications by abstracting the behavior of the database and the data stored by the database. Whether you are writing a single-tiered, two-tiered, or multi-tiered application, you can isolate your user interface from the data access layer as illustrated in Figure 12.1.

Figure 12.1 User-interface to dataset connections in all database applications



A form represents the user interface, and contains data controls and other user interface elements. The data controls in the user interface connect to datasets which represent information from the tables in the database. A data source links the data controls to these datasets. By isolating the data source and datasets in a data module, the form can remain unchanged as you scale your application up. Only the datasets must change.

Note Some user interface elements require special attention when planning for scalability. For example, different databases enforce security in different ways. See “Database security” on page 12-3 for more information on handling user authentication in a uniform manner as you change databases.

When using Delphi’s data access components (whether they use the BDE, ADO, or InterBase Express) it is easy to scale from one-tiered to two-tiered. Only a few properties on the dataset must change to direct the dataset to connect to an SQL server rather than a local database.

A flat-file database application is easily scaled to the client in a multi-tiered application because both architectures use the same client dataset component. In fact, you can write an application that acts as both a flat-file application and a multi-tiered client (see “Using the briefcase model” on page 13-17).

If you plan to scale your application up to a three-tiered architecture eventually, you can write your one- or two-tiered application with that goal in mind. In addition to isolating the user interface, isolate all logic that will eventually reside on the middle tier so that it is easy to replace at a later time. You can even connect your user interface elements to client datasets (used in multi-tiered applications), and connect them to local versions of the InterBase, BDE- or ADO- enabled datasets in a separate data module that will eventually move to the middle tier. If you do not want to introduce this artifice of an extra dataset layer in your one- and two-tiered

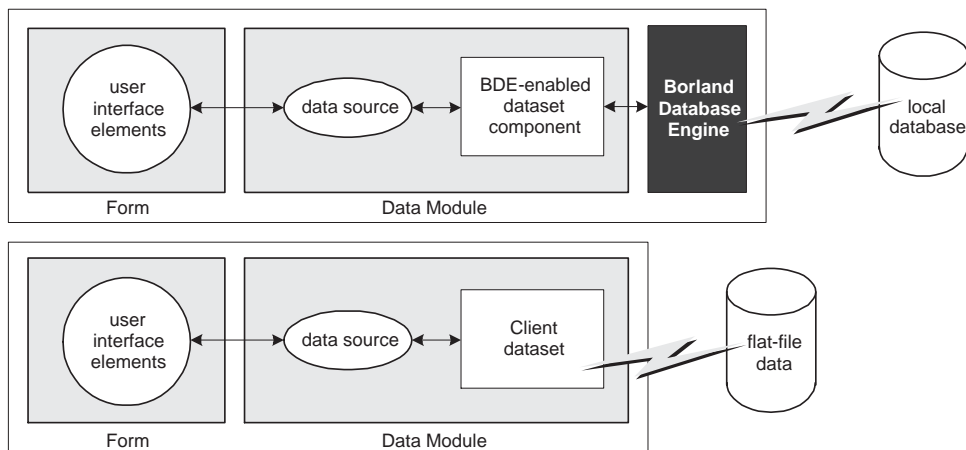
applications, it is still easy to scale up to a three-tiered application at a later date. See “Scaling up to a three-tiered application” on page 13-18 for more information.

Single-tiered database applications

In single-tiered database applications, the application and the database share a single file system. They use local databases or files that store database information in a flat-file format.

A single application comprises the user interface and incorporates the data access mechanism (either the BDE or a system for loading and saving flat-file database information). The type of dataset component used to represent database tables depends on whether the data is stored in a local database (such as Paradox, dBASE, Access, or FoxPro) or in a flat file. Figure 12.2 illustrates these two possibilities:

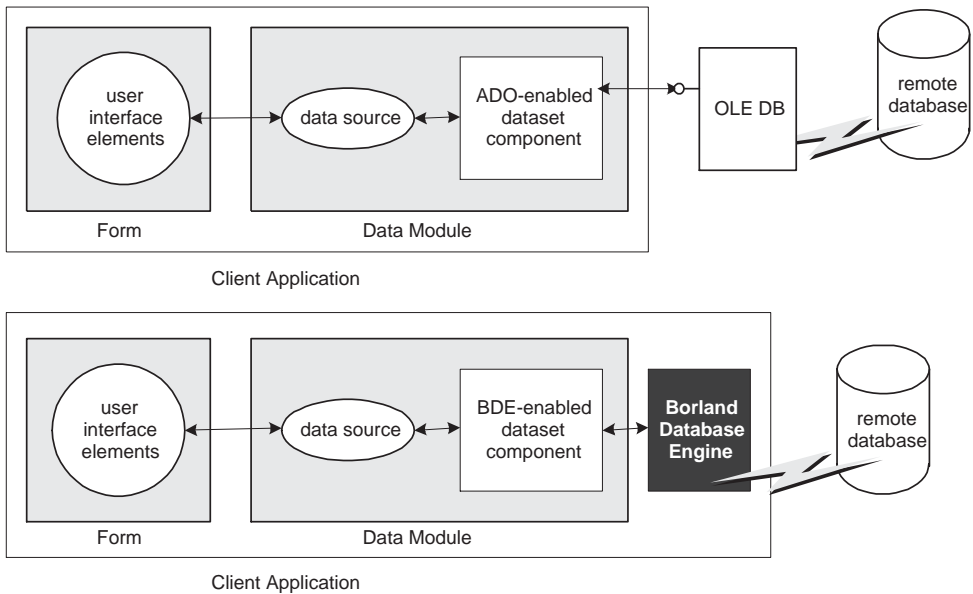
Figure 12.2 Single-tiered database application architectures



For more information on building single-tiered database applications, see Chapter 13, “Building one- and two-tiered applications”.

Two-tiered database applications

In two-tiered database applications, a client application provides a user interface to data, and interacts directly with a remote database server. Figure 12.3 illustrates this relationship.

Figure 12.3 Two-tiered database application architectures

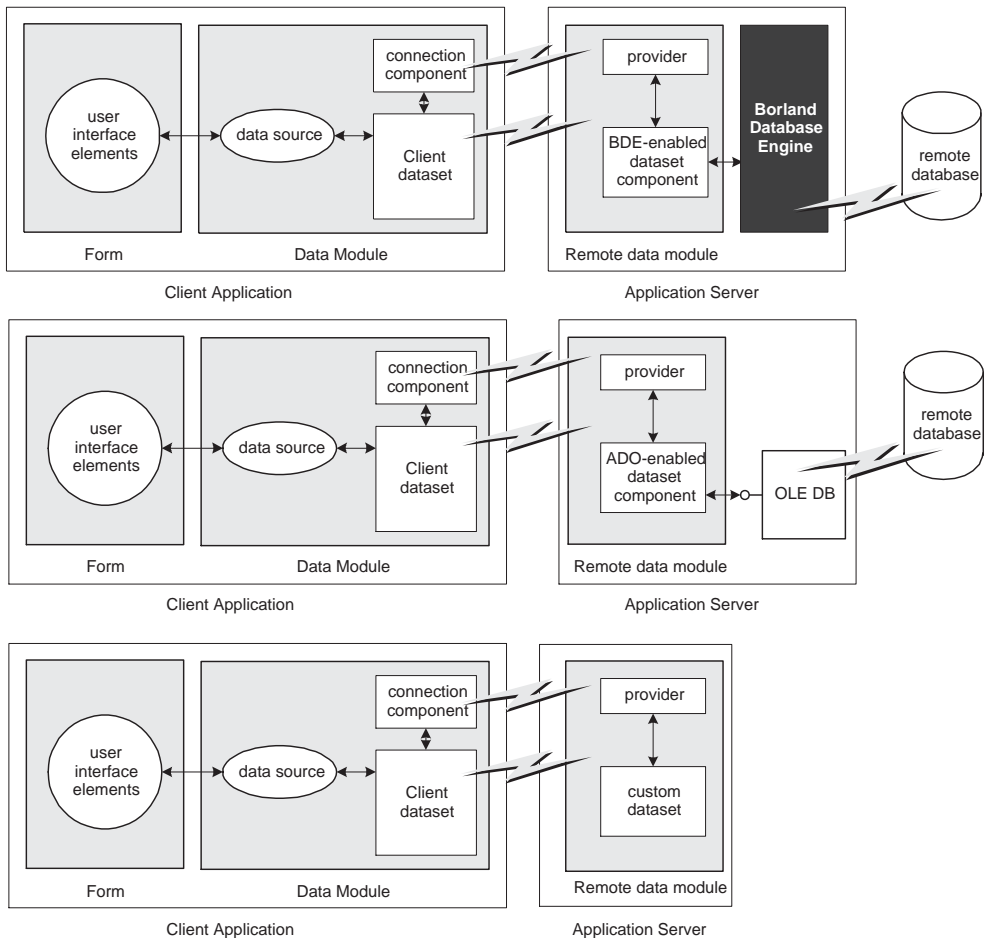
In this model, all applications are database clients. A *client* requests information from and sends information to a database server. A server can process requests from many clients simultaneously, coordinating access to and updating of data.

For more information on building two-tiered database applications, see “BDE-based applications” on page 13-2 and “ADO-based applications” on page 13-10.

Multi-tiered database applications

In multi-tiered database applications, an application is partitioned into pieces that reside on different machines. A client application provides a user interface to data. It passes all data requests and updates through an application server (also called a “remote data broker”). The application server, in turn, communicates directly with a remote database server or some other custom dataset. Usually, in this model, the client application, the application server, and the remote database server are on separate machines. Figure 12.4 illustrates these relationships for different types of multi-tiered applications.

Figure 12.4 Multi-tiered database architectures



You can use Delphi to create both client applications and application servers. The client application uses standard data-aware controls connected through a data source to one or more client dataset components in order to display data for viewing and editing. Each client dataset communicates with an application server through an *AppServer* interface that is implemented by the application server's remote data module. The client application can use a variety of protocols (TCP/IP, HTTP, DCOM, MTS, or CORBA) to establish this communication. The protocol depends on the type of connection component used in the client application and the type of remote data module used in the server application.

The application server contains provider components that mediate the communication between client datasets on the client application and the datasets on the application server. All data is passed between the client application and the provider components through the *AppServer* interface.

Usually, several client applications communicate with a single application server in the multi-tiered model. The application server provides a gateway to your databases for all your client applications, and it lets you provide enterprise-wide database tasks in a central location, accessible to all your clients. For more information about creating and using a multi-tiered database application, see Chapter 14, “Creating multi-tiered applications.”

Designing the user interface

The Data Controls page of the Component 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 Chapter 26, “Using data controls”.

Data-aware controls get data from and send data to a data source component (*TDataSource*). A data source component acts as a conduit between the user interface and a dataset component which represents a set of information from the tables in 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. An application’s data source components usually reside in a data module, separate from the data-aware controls on forms.

The data-aware controls you add to your user interface depend on what type of data you are displaying (plain text, formatted text, graphics, multimedia elements, and so on). In addition, your choice of controls is determined by how you want to organize the information and how (or if) you want to let users navigate through the records of datasets and add or edit data.

The following sections introduce the components you can use for various types of user interface.

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 page of the Component palette provides a wide selection of controls to represent different kinds of fields. For more information about specific data-aware controls, see “Controls that represent a single field” on page 26-8.

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" on page 26-16 and "Creating a grid that contains other data-aware controls" on page 26-27.

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 forms" on page 20-24 or "Working with nested tables" on page 20-26.
- **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. While the result can sometimes be effective, it is usually confusing for users to understand the data relationships.

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 page of the Component 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 page on the Component 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 Chapter 27, "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. For more information about using maintained aggregates, see “Using maintained aggregates” on page 24-9.

Selecting what data to show

Often, the data you want to surface in your database application does not correspond exactly to the data in a single database table. You may want to use only a subset of the fields or a subset of the records in a table. You may want to combine the information from more than one table into a single joined view.

The data available to your database application is controlled by your choice of dataset component. Datasets abstract the properties and methods of a database table, so that you do not need to make major alterations depending on whether the data is stored in a database table or derived from one or more tables in the database. For more information on the common properties and methods of datasets, see Chapter 18, “Understanding datasets”.

Your application can contain more than one dataset. Each dataset represents a logical table. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. You might need to alter the type of dataset component, or the way it specifies the data it contains, but the rest of your user interface can continue to work without alteration.

When using the BDE to access your data, you can use any of the following types of dataset:

- **Table components (TTable):** Tables correspond directly to the underlying tables in the database. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. Tables are described in more detail in Chapter 20, “Working with tables”. Persistent fields are described in “Persistent field components” on page 19-4. Ranges and filters are described in “Working with a subset of data” on page 20-11.
- **Query components (TQuery):** Queries provide the most general mechanism for specifying what appears in a BDE-based dataset. You can combine the data from multiple tables using joins, and limit the fields and records that appear based on any criteria you can express in SQL. For more information on queries, see Chapter 21, “Working with queries”.
- **Stored procedures (TStoredProc):** Stored procedures are sets of SQL statements that are named and stored on an SQL server. If your database server defines a stored procedure that returns the dataset you want, you can use a stored procedure component. For more information on stored procedures, see Chapter 22, “Working with stored procedures”.
- **Nested datasets (TNestedTable):** Nested datasets represent the records in an Oracle8 nested detail set. Delphi does not let you create Oracle8 tables with nested dataset fields, but you can edit and display data from existing dataset fields using nested datasets. The nested dataset gets its data from a dataset field component in

a dataset which contains Oracle8 data. See “Working with nested tables” on page 20-26 and “Working with dataset fields” on page 19-26 for more information on using nested datasets to represent dataset fields.

When using ADO to access your data, you can use any of the following types of dataset:

- **ADO datasets (TADODataset):** ADO datasets provide the most flexible mechanism for accessing data using ADO. ADO datasets can represent a single database table or the results of an SQL query. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. You can specify an SQL statement that generates the data. ADO datasets are described in more detail in “Features common to all ADO dataset components” on page 23-12 and “Using TADODataset” on page 23-18.
- **ADO table components (TADOTable):** ADO tables correspond directly to the underlying tables in the database. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. ADO tables are described in more detail in “Using TADOTable” on page 23-19.
- **ADO query components (TADOQuery):** ADO Queries represent the result set from running an SQL command or data definition language (DDL) statement. For more information on ADO queries, see “Using TADOQuery” on page 23-20.
- **ADO stored procedures (TADOStoredProc):** If your database server defines a stored procedure that returns the dataset you want, you can use an ADO stored procedure component. For more information on ADO stored procedures, see “Using TADOStoredProc” on page 23-22.

If you are using InterBase for your database server, you can use any of the following types of dataset:

- **IB datasets (TIBDataSet):** IB datasets represents the result set of an SQL statement (usually a SELECT statement). You can specify SQL statements for selecting and updating the data buffered by the dataset.
- **IB table components (TIBTable):** IB tables get their data directly from an InterBase table or view. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using filters.
- **IB query components (TIBQuery):** IB queries represent the result set from running an SQL command. IB queries are the most easily scaled dataset component when moving from local InterBase to a remote InterBase server.
- **IB stored procedures (TIBStoredProc):** IBStoredProc executes an InterBase Execute stored procedure. These datasets do not return a result set: for stored procedures that return a result set you must use *TIBDataSet* or *TIBQuery*.

If you are not using the BDE, ADO, or InterBase, Delphi provides the following options:

- **Client datasets (TClientDataSet):** Client datasets cache the records of the logical dataset in memory. Because of that, they can only hold a limited number of records. Client datasets are populated with data in one of two ways: from an application server or from flat-file data stored on disk. They do not require a database engine such as BDE or ADO, but rely on a single DLL (Midas.dll). For more information about client datasets, see Chapter 24, “Creating and using a client dataset”.
- **Custom datasets:** You can create your own custom descendants of *TDataSet* to represent a body of data that you create or access in code you write. 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 Chapter 31, “Overview of component creation”.

Writing reports

If you want to let your users print database information from the datasets in your application, you can use the report components on the QReport page of the Component palette. Using these components you can visually build banded reports to present and summarize the information in your database tables. You can add summaries to group headers or footers to analyze the data based on grouping criteria.

Start a report for your application by selecting the QuickReport icon from the New Items dialog. Select File | New from the main menu, and go to the page labeled Business. Double-click the QuickReport Wizard icon to launch the wizard.

Note See the QuickReport demo that ships with Delphi for an example of how to use the components on the QReport page.

Building one- and two-tiered applications

One- and two-tiered applications include the logic that manipulates database information in the same application that implements the user interface. Because the data manipulation logic is not isolated in a separate tier, these types of applications are most appropriate when there are no other applications sharing the same database information. Even when other applications share the database information, these types of applications are appropriate if the database is very simple, and there are no data semantics that must be duplicated by all applications that use the data.

You may want to start by writing a one- or two-tiered application, even when you intend to eventually scale up to a multi-tiered model as your needs increase. This approach lets you avoid having to develop data manipulation logic up front so that the application server can be available while you are writing the user interface. It also allows you to develop a simpler, cheaper prototype before investing in a large, multi-system development project. If you intend to eventually scale up to a multi-tiered application, you can isolate the data manipulation logic so that it is easy to move it to a middle tier at a later date.

Delphi provides support for two types of single-tiered applications: applications that use a local database (such as Paradox, dBase, Access, or Local Interbase) and flat-file database applications. Two-tiered applications use a driver to access a remote database.

The considerations when writing single-tiered applications that use a local database and two-tiered applications are essentially the same, and depend primarily on the mechanism you choose to connect to the database. Delphi provides three different built-in mechanisms for these types of applications:

- BDE-based applications
- ADO-based applications
- InterBaseExpress applications

Flat file database applications are based on the support for client datasets included in MIDAS.DLL.

BDE-based applications

Because the data access components (and Borland Database Engine) handle the details of reading data, updating data, and navigating data, writing BDE-based two-tiered applications is essentially the same as writing BDE-based one-tiered applications.

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 many advantages. BDE-based applications allow you to use the powerful library of Borland Database Engine API calls. Even if you do not want to use the BDE API, writing BDE-based applications gives you support for the following features not available to other applications such as flat-file database application:

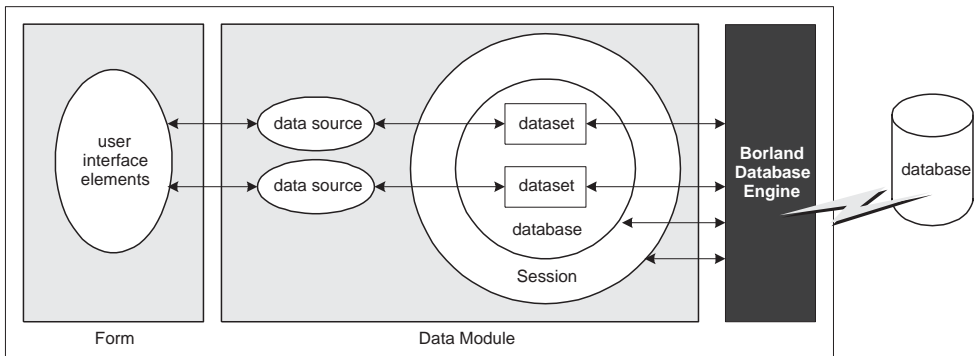
- Connecting to databases.
- Using transactions.
- Caching updates.
- Creating and restructuring database tables.

BDE-based architecture

A BDE-based one- or two-tiered application includes

- A user interface containing data-aware controls.
- One or more datasets that represent information from the database tables.
- A datasource component for each dataset to connect the data-aware controls to the datasets.
- Optionally, one or more database components to control transactions in both one- and two-tiered applications and to manage database connections in two-tiered applications.
- Optionally, one or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between these elements is illustrated in Figure 13.1:

Figure 13.1 Components in a BDE-based application

Understanding databases and datasets

Databases contain information stored in tables. They may also include tables of information about what is contained in the database, objects such as indexes that are used by tables, and SQL objects such as stored procedures. See Chapter 17, “Connecting to databases” for more information about databases.

The Data Access page of the Component palette contains various dataset components that represent the tables contained in a database or logical tables constructed out of data stored in those database tables. See “Selecting what data to show” on page 12-13 for more information about these dataset components. You must include a dataset component in your application to work with database information.

Each BDE-enabled dataset component on the Data Access page has a published *DatabaseName* property that specifies the database which contains the table or tables that hold the information in that dataset. When setting up your application, you must use this property to specify the database before you can bind the dataset to specific information contained in that database. What value you specify depends on whether

- 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). Use the BDE Administration tool or the SQL explorer to create and manage BDE aliases.
- The database is a Paradox or dBASE database. If you are using a Paradox or dBASE database, *DatabaseName* can specify the directory where the database tables are located.
- You are using explicit database components. Database components (*TDatabase*) represent a database in your application. If you don’t add a database component explicitly, a temporary one is created for you automatically, based on the value of the *DatabaseName* property. If you are using explicit database components, *DatabaseName* is the value of the *DatabaseName* property of the database component. See “Understanding persistent and temporary database components” on page 17-1 for more information about using database components.

Using sessions

Sessions isolate data access operations such as database connections, and manage groups of databases. All use of the Borland Database Engine takes place in the context of a session. You can use sessions to specify configuration information that applies to all the databases in the session. This allows you to override the default behavior specified using the BDE administration tool.

You can use a session to

- Manage BDE aliases. You can create new aliases, delete aliases, and modify existing aliases. By default, changes affect only the session, but you can write changes so that they are added to the permanent BDE configuration file. For more information on managing BDE aliases, see “Working with BDE aliases” on page 16-9.
- Control when database connections in two-tiered applications are closed. Keeping database connections open when none of the datasets in the database are active ties up resources that could be released, but improves speed and reduces network traffic. To keep database connections open even when there are no active datasets, the *KeepConnections* property should be *True* (the default).
- Manage access to password-protected Paradox and dBASE files in one-tiered applications. Datasets that access password-protected Paradox and dBASE tables use the session component to supply a password when these tables must be opened. You can override the default behavior (a password dialog that appears whenever a password is needed), to supply passwords programmatically. If you intend to scale your one-tiered application to a two-tiered or multi-tiered application, you can create a common user interface for obtaining user authentication information that need not change when you switch to using remote database servers which require a username and password at the server (rather than table) level. For more information about using sessions to manage Paradox and dBASE passwords, see “Working with password-protected Paradox and dBase tables” on page 16-13.
- Specify the location of special Paradox directories. Paradox databases that are shared on a network use a net directory which contains temporary files that specify table and record locking information. Paradox databases also use a private directory where temporary files such as the results of queries are kept. For more information on specifying these directory locations, see “Specifying Paradox directory locations” on page 16-13.

If your application may be accessing the same database multiple times simultaneously, you must use multiple sessions to isolate these uses of the database. Failure to do so will disrupt the logic governing transactions on that database (including transactions created for you automatically). Applications risk simultaneous access when running concurrent queries or when using multiple threads. For more information about using multiple sessions, see “Managing multiple sessions” on page 16-16.

Unless you need to use multiple sessions, you can use the default session.

Connecting to databases

The Borland Database Engine includes drivers to connect to different databases. The Standard version of Delphi includes only the drivers for local databases: Paradox, dBASE, FoxPro, and Access. With the Professional version, you also get an ODBC adapter that allows the BDE to use ODBC drivers. By supplying an ODBC driver, your application can use any ODBC-compliant database. Some versions also include drivers for remote database servers. Use the drivers installed with SQL Links to communicate with remote database servers such as InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

Note The only difference between a BDE-based one-tiered application and a BDE-based two-tiered application is whether it uses local databases or remote database servers.

Using 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.

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, such as the transfer of funds described previously.

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.

Note You can also minimize the number of transactions you need by caching updates. For more information about cached updates, see Chapter 25, "Working with cached updates."

Explicitly controlling transactions

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the methods and properties of the database component, such as *StartTransaction*, *Commit*, *Rollback*, *InTransaction*, and *TransIsolation*. The main advantage to using the methods and properties of a database component to control transactions is that it provides a clean, portable application that is not dependent on a particular database or server.
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. For more information about query components, see Chapter 21, “Working with queries.” 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.

One-tiered applications can’t use passthrough SQL. You can use the database component to create explicit transactions for local databases. However, there are limitations to using local transactions. For more information on using local transactions, see “Using local transactions” on page 13-9.

When writing two-tiered applications (which require SQL links), you can use either a database component or passthrough SQL to manage transactions. For more information about using passthrough SQL, see “Using passthrough SQL” on page 13-8.

Using a database component for transactions

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. 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.

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 your changes.

When using a database component, you code a single transaction as follows:

- 1 Start the transaction by calling the database’s *StartTransaction* method:

```
DatabaseInterBase.StartTransaction;
```

- 2 Once the transaction is started, all subsequent database actions are considered part of the transaction until the transaction is explicitly terminated. You can determine whether a transaction is in process by checking the database component’s *InTransaction* property. While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level. For more information about transaction isolation levels, see “Using the *TransIsolation* property” on page 13-7.

- 3 When the actions that make up the transaction have all succeeded, you can make the database changes permanent by using the database component's *Commit* method:

```
DatabaseInterBase.Commit;
```

Commit is usually attempted in a **try...except** statement. That way, if a transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

- 4 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. To discard these changes, use the database component's *Rollback* method:

```
DatabaseInterBase.Rollback;
```

Rollback usually occurs in

- Exception handling code when you cannot recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Using the TransIsolation property

TransIsolation specifies the *transaction isolation level* for a database component's transactions. 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.

The default setting for *TransIsolation* is *tiReadCommitted*. The following table summarizes possible values for *TransIsolation* and describes what they mean:

Table 13.1 Possible values for the *TransIsolation* property

Isolation level	Meaning
<i>tiDirtyRead</i>	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
<i>tiReadCommitted</i>	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
<i>tiRepeatableRead</i>	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. 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.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, the BDE uses the next highest isolation level. The actual isolation level used by some servers is shown in Table 13.2,

“Transaction isolation levels.” For a detailed description of how each isolation level is implemented, see your server documentation.

Table 13.2 Transaction isolation levels

Server	Specified Level	Actual Level
Oracle	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted tiRepeatableRead (READONLY)
Sybase, MS-SQL	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted Not supported
DB2	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead tiReadCommitted tiRepeatableRead
Informix	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead tiReadCommitted tiRepeatableRead
InterBase	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted tiRepeatableRead
Paradox, dBASE, Access, FoxPro	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead Not supported Not supported

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.

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

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 correctly. 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. For more information about SQLPASSTHRU modes, see the help file for the BDE Administration utility.

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 local 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.

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.
- 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.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- 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.

Caching updates

The Borland Database Engine provides support for caching updates. When you cache updates, your application retrieves data from a database, makes all changes to a local, cached copy of the data, and applies the cached changes to the dataset as a unit. Cached updates are applied to the database in a single transaction.

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.

You can tell BDE-enabled datasets to cache updates using the *CachedUpdates* property. When the changes are complete, they can be applied by the dataset component, by the database component, or by a special update object. When changes can't be applied to the database without additional processing (for example, when working with a joined query), you must use the *OnUpdateRecord* event to write changes to each table that makes up the joined view.

For more information on caching updates, see Chapter 25, "Working with cached updates".

Note If you are caching updates, you may want to consider moving to a multitiered model to have greater control over the application of updates. For more information about the multitiered model, see Chapter 14, "Creating multi-tiered applications".

Creating and restructuring database tables

In BDE-based applications, you can use the *TTable* component to create new database tables and to add indexes to existing tables.

You can create tables either at design time, in the Forms Designer, or at runtime. To create a table, you must specify the fields in the table using the *FieldDefs* property, add any indexes using the *IndexDefs* property, and call the *CreateTable* method (or select the Create Table command from the table's context menu). For more detailed instructions on creating tables, see "Creating a table" on page 20-17.

Note When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, reference fields, and dataset fields).

If you want to restructure a table at runtime (other than by adding indexes), you must use the BDE API *DbiDoRestructure*. You can add indexes to an existing table using the *AddIndex* method of *TTable*.

Note At design time, you can use the Database Desktop to create and restructure Paradox and dBASE tables. To create and restructure tables on remote servers, use the SQL Explorer and restructure the table using SQL.

ADO-based applications

Delphi applications that use the ADO components for data access can be either one- or two-tier. Which category an application falls under is predicated on the database type used. For instance, using ADO to access a Microsoft SQL Server database will always be a two-tier application because SQL Server is an SQL database system. SQL database systems are typically located on a dedicated SQL server. On the other hand,

an application that uses ADO to access some local database type, like dBASE or FoxPro, will always be a one-tier application.

There are four major areas involved in database access using ADO and the Delphi ADO components. These areas of concern are the same regardless of whether the application is one- or two-tier. These five concerns are:

- ADO-based architecture
- Connecting to ADO databases
- Retrieving data
- Creating and restructuring ADO database tables

ADO-based architecture

An ADO-based application includes the following functional areas

- A user interface with visual data-aware controls. Visual data controls are optional if all data access is done programmatically.
- One or more dataset components that represent information from tables or queries.
- One datasource component for each dataset component to act as the conduit between dataset component and one or more visual data-aware controls.
- A connection component to connect to the ADO data store. The connection component acts as a conduit between the application's dataset components and the database accessed through the data store.

The ADO layer of an ADO-based Delphi 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 external entities must be present and accessible to the ADO-based application for it to be fully functional.

Understanding ADO databases and datasets

The ADO page of the Component Palette contains all of the components necessary for connecting to databases and for accessing the tables in them.

All of the metadata objects an ADO-based application are contained in the database accessed through the ADO data store. To access these objects or the data stored in them, an application must first connect to the data store. See "Connecting to ADO data stores" on page 23-2 for information on connecting to data stores.

The data of a database is stored in one or more tables. You must include at least one ADO dataset component (*TADODataSet*, *TADOQuery*, and so on) in your application to work with the data stored in a database's tables. See "Retrieving data" on page 13-12 and "Using ADO datasets" on page 23-11 for more information on using ADO dataset components to access data in tables.

Connecting to ADO databases

An ADO-based application Delphi uses ADO 2.1 to interact with an OLE DB provider to connect to a data store and access its data. One of the things 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.

The provider can represent one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must also 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 the application with the data store, the ADO connection component is configured to use one of the available providers. Once the connection component is connected to the data store, dataset components can be associated with the connection component to access the tables in the database. See “Connecting to ADO data stores” on page 23-2 for information on connecting to data stores.

In addition to providing an application’s access to the database, the connection component encapsulates the ADO transaction processing capabilities.

Retrieving data

Once an application has established a valid connection to a database, dataset components can be used to access data in the database. The ADO page of the Component Palette contains the ADO dataset components needed to access data from ADO data stores.

These components include *TADODataset*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. All of these components are capable of retrieving data from ADO data stores, programmatically modifying the data, and presenting the data to an application’s user for interactive use of the data. For more information on using the ADO dataset components to retrieve and modify data, see “Using ADO datasets” on page 23-11.

To make the data accessed with an ADO dataset component visually accessible in an application, use the stock data-aware controls. There are no ADO-specific data-aware controls. For details on using data-aware controls, see “Using common data control features” on page 26-1.

The standard data source component is used as a conduit between the ADO dataset components and the data-aware controls. There is no dedicated data source component for ADO. For details on using data source components, see “Using data sources” on page 26-5.

Where needed, persistent field objects can be used to represent fields in the ADO dataset components. As with the data-aware controls and data source components, simply use the inherent Delphi field classes (*TField* and descendents). For details on using dynamic and persistent field objects, see “Understanding field components” on page 19-2.

Creating and restructuring ADO database tables

Creating and deleting metadata in an ADO database from a Delphi application must be done using SQL. Similarly, restructuring tables is done using SQL statements. Changing other metadata objects cannot be done per se. Instead, you need to delete the metadata object and then replace it with a new one with different attributes.

There are many database types that can be accessed through ADO and not all of the drivers for specific database types support all of the same SQL syntax. It is beyond the scope of this document to describe all of the SQL syntax supported by each database type and all of 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 database system.

In general, use the CREATE TABLE statement to create tables in the 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, and CREATE SCHEMA.

For each of the CREATE statements, there is a corresponding DROP statement to delete a metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, and DROP SCHEMA.

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.

Issue these metadata statements from the ADO command or the ADO query component. For details on using the ADO command component to execute commands, see “Executing commands” on page 23-26.

Flat-file database applications

Flat-file database applications are single-tiered applications that use *TClientDataSet* to represent all of their datasets. The client dataset holds all its data in memory, which means that this type of application is not appropriate for extremely large datasets.

Flat-file database applications do not require the Borland Database Engine (BDE) or ActiveX Data Objects (ADO). Instead, they only use MIDAS.DLL. By using only MIDAS.DLL, flat-file applications are easier to deploy because you do not need to install, configure, and maintain software that manages database connections.

Because these applications do not use a database, there is no support for multiple users. Instead, the datasets are dedicated entirely to the application. Data can be saved to flat 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.

Client datasets (located on the MIDAS page of the Component palette) form the basis of flat-file database applications. They provide support for most of the database operations you perform with other datasets. You use the same data-aware controls and data source components that you would use in a BDE-based single-tiered application. You don't use database components, because there is no database connection to manage, and no transactions to support. You do not need to be concerned with session components unless your application is multi-threaded. For more information about using client datasets, see Chapter 24, "Creating and using a client dataset".

The main differences in writing flat-file database applications and other single-tiered database applications lie in how you create the datasets and how you load and save data.

Creating the datasets

Because flat-file database applications do not use existing databases, you are responsible for creating the datasets yourself. 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. However, indexes are not saved with the table. You need to recreate them every time you load the table.

When beginning a flat-file database application, you may want to first create and save empty files for your datasets before beginning the writing of the application itself. This way, you do not need to define the metadata for your client datasets in the final application.

How you create your client dataset depends on whether you are creating an entirely new dataset, or converting an existing BDE-based application.

Creating a new dataset using persistent fields

The following steps describe how to create a new client dataset using the Fields Editor:

- 1 From the MIDAS page of the Component palette, add a *TClientDataSet* component to your application.
- 2 Right-click the client dataset and select Fields Editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field definition. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.

Continue adding fields in the fields editor until you have described your client dataset.

- 3 Right-click the client dataset and choose Create DataSet. This creates an empty client dataset from the persistent fields you added in the Fields Editor.

- 4 Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)
- 5 In the File Save dialog, choose a file name and save a flat file copy of your client dataset.

Note You can also create the client dataset at runtime using persistent fields that are saved with the client dataset. Simply call the *CreateDataSet* method.

Creating a dataset using field and index definitions

Creating a client dataset using field and index definitions is much like using a *TTable* component to create a database table. There is no *DatabaseName*, *TableName*, or *TableType* property to specify, as these are not relevant to client datasets. However, just as with *TTable*, you use the *FieldDefs* property to specify the fields in your table and the *IndexDefs* property to specify any indexes. Once the table is specified, right-click the client dataset and choose Create DataSet at design time, or call the *CreateDataSet* method at runtime.

When defining the index definitions for your client dataset, two properties of the index definition apply uniquely to client datasets. These are *TIndexDef.DescFields* and *TIndexDef.CaseInsFields*.

DescFields lets you define indexes that sort records in ascending order on some fields and descending order on other fields. Instead of using the *ixDescending* option to sort in descending order on all the fields in the index, list only those fields that should sort in descending order as the value of *DescFields*. For example, when defining an index that sorts on Field1, then Field2, then Field3, setting *DescFields* to

```
Field1;Field3
```

results in an index that sorts Field2 in ascending order and Field1 and Field3 in descending order.

CaseInsFields lets you define indexes that sort records case-sensitively on some fields and case-insensitively on other fields. Instead of using the *isCaseInsensitive* option to sort case-insensitively on all the fields in the index, list only those fields that should sort case-insensitively as the value of *CaseInsFields*. Like *DescFields*, *CaseInsFields* takes a semicolon-delimited list of field names.

You can specify the field and index definitions at design time using the Collection editor. Just choose the appropriate property in the Object Inspector (*FieldDefs* or *IndexDefs*), and double-click to display the Collection editor. Use the Collection editor to add, delete, and rearrange definitions. By selecting definitions in the Collection editor you can edit their properties in the Object Inspector.

You can also specify the field and index definitions in code at runtime. For example, the following code creates and activates a client dataset in the form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with ClientDataSet1 do
  begin
    with FieldDefs.AddFieldDef do
```

```

begin
  DataType := ftInteger;
  Name := 'Field1';
end;
with FieldDefs.AddFieldDef do
begin
  DataType := ftString;
  Size := 10;
  Name := 'Field2';
end;
with IndexDefs.AddIndexDef do
begin
  Fields := 'Field1';
  Name := 'IntIndex';
end;
CreateDataSet;
end;
end;

```

Creating a dataset based on an existing table

If you are converting an existing BDE-based application into a single-tiered flat-file application, you can copy existing tables and save them as flat-file tables from the IDE. The following steps indicate how to copy an existing table:

- 1 From the Data Access page of the Component palette, add a *TTable* component to your application. Set its *DatabaseName* and *TableName* properties to identify the existing database table. Set its *Active* property to *True*.
- 2 From the MIDAS page of the Component palette, add a *TClientDataSet* component.
- 3 Right-click the client dataset and select Assign Local Data. In the dialog that appears, choose the table component that you added in step 1. Choose OK.
- 4 Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)
- 5 In the File Save dialog, choose a file name and save a flat-file copy of your database table.

Loading and saving data

In flat-file database applications, all modifications to the table exist only in an in-memory change log. This log is 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. For more information about the change log, see “Editing data” on page 24-4.

Even when you have merged changes into the data of the client dataset, this data still exists only in memory. While it will persist 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, 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.

When you save a client dataset, the metadata that describes the record structure is saved with the dataset, but not the indexes. Because of this, you may want to add code that recreates the indexes when you load the data from file. Alternately, you might want to write your application so that it always creates indexes on the fly in an as-needed fashion.

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.

Using the briefcase model

Most of this section has described creating and using a client dataset in a one-tiered application. The one-tiered model can be combined with a multi-tiered model to create what is called the briefcase model.

Note The briefcase model is sometimes called the disconnected model, or mobile computing.

When operating on site, a briefcase model application looks like a multi-tiered model: a user starts a client application on one machine and connects over a network to an application server on a remote machine. The client requests data from the application server, and sends updates to it. The updates are applied by the application server to a database that is presumably shared with other clients throughout an organization.

Suppose, however, that your onsite company database contains valuable customer contact data that your sales representatives can use and update in the field. In this case, it would be useful if your sales reps could download some or all of the data from the company database, 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 reps return onsite, they need to upload their data changes to the company database for everyone to use. This ability to work with data off-line and then apply updates online at a later date is known as the “briefcase” model.

By using the briefcase model, you can take advantage of the client dataset component’s ability to read and write data to flat files to create client applications that can be used both online with an application server, and off-line, as temporary one-tiered applications.

To implement the briefcase model, you must

- 1 Create a multi-tiered server application as described in “Creating the application server” on page 14-11.
- 2 Create a flat-file database application as your client application. Add a connection component and set the *RemoteServer* property of your client datasets to specify this connection component. This allows them to talk to the application server created in step 1. For more information about connection components, see “Connecting to the application server” on page 14-19.
- 3 In the client application, try on start-up to connect to the application server. If the connection fails, prompt the user for a file and read in the local copy of the data.
- 4 In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see “Updating records” on page 24-20.

Scaling up to a three-tiered application

In a two-tiered client/server application, the application is a client that talks directly to a database server. Even so, the application can be thought of as having two parts: a database connection and a user interface. To make a two-tiered client/server application into a multi-tiered application you must:

- Split your existing application into an application server that handles the database connection, and a client application that contains the user interface.
- Add an interface between the client and the application server.

There are a number of ways to proceed, but the following sequential steps may best keep your translation work to a minimum:

- 1 Create a new project for the application server, starting with a remote data module. See “Creating the application server” on page 14-11 for details on how to do this.
- 2 Duplicate the relevant database connection portions of your former two-tiered application, and for each dataset, add a provider component that will act as a data conduit between the application server and the client. For more information on using a provider component, see Chapter 15, “Using provider components”.
- 3 Copy your existing two-tiered project, remove its direct database connections, add an appropriate connection component to it. For more information about creating and using connection components, see “Connecting to the application server” on page 14-19.
- 4 Substitute a client dataset for each BDE-enabled or ADO-enabled dataset component in the original project. For general information about using a client dataset component, see Chapter 24, “Creating and using a client dataset.”

- 5 In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see “Updating records” on page 24-20.
- 6 Move the dataset components to the application server’s data modules. Set the *DataSet* property of each provider to specify the corresponding datasets. For more information about linking a dataset to a provider component, see Chapter 15, “Using provider components”.

Creating multi-tiered applications

This chapter describes how to create a multi-tiered, client/server database application. A multi-tiered client/server application is partitioned into logical units 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.” With Delphi 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 platforms not directly supported by Delphi.

Delphi support for multi-tiered applications is based on the Multi-tier Distributed Application Services Suite (MIDAS). This chapter focuses on creating a three-tiered database application using the MIDAS technology. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

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 remote 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 Borland Database Engine). 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, CORBA, or MTS, you can take advantage of the security models they support.

Understanding MIDAS technology

MIDAS provides the mechanism by which client applications and application servers communicate database information. Using MIDAS requires MIDAS.DLL, which is used by both client and server applications to manage datasets stored as data packets. Building MIDAS applications may also require the SQL explorer to help in database administration and to import server constraints into the Data Dictionary so that they can be checked at any level of the multi-tiered application.

Note You must purchase server licenses for deploying your MIDAS applications.

MIDAS-based multi-tiered applications use the components on the MIDAS page of the component palette, plus a remote data module that is created by a wizard on the

Multitier page of the New Items dialog. These components are described in Table 14.1:

Table 14.1 MIDAS components

Component	Description
remote data modules	Specialized data modules that can act as a COM Automation server or CORBA server 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 to manage data stored in data packets.
connection components	A family of components that locate the server, form connections, and make the <i>IAppServer</i> interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

Overview of a MIDAS-based multi-tiered application

The following numbered steps illustrate a normal sequence of events for a MIDAS-based multi-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 from 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, about data constraints imposed by the database) 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 a record can’t be posted to the server (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 traditional two-tiered application that uses cached updates. Structurally, the client application looks a lot like a flat-file single-tiered application. User interaction takes place through standard data-aware controls that display data from a client dataset component. For detailed information about using the properties, events, and methods of client datasets, see Chapter 24, “Creating and using a client dataset.”

Unlike in a flat-file application, the client dataset in a multi-tiered application obtains its data through the *IAppServer* interface on the application server. It uses this interface to post updates to the application server as well. For more information about the *IAppServer* interface, see “Using the IAppServer interface” on page 14-7. The client 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:

Table 14.2 Connection components

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TOLEnterpriseConnection	OLEEnterprise (RPCs)
TCorbaConnection	CORBA (IIOP)

Note Two other connection components, *TRemoteServer* and *TMIDASConnection*, are provided for backward compatibility.

For more information about using connection components, see “Connecting to the application server” on page 14-19.

The structure of the application server

The application server includes a remote data module that provides an *IAppServer* interface, which client applications use to communicate with data providers. 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 OLEEnterprise to connect to the application server, unless you want to install the application server with MTS.
- **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 MTS. You can use MTS remote data modules with DCOM, HTTP, Sockets, or OLEEnterprise.
- **TCorbaDataModule:** This is a CORBA server. Use this type of remote data module to provide data to CORBA clients.

As with any data module, you can include any nonvisual component in the remote data module. In addition, the remote data module includes a dataset provider component for each dataset the application server makes available to client applications. A dataset provider

- Receives data requests from the client, fetches the requested data from the database server, packages the data for transmission, and sends the data to the client dataset. This activity is called “providing.”
- Receives updated data from the client dataset, applies updates to the database or source dataset, and logs any updates that cannot be applied, returning unresolved updates to the client for further reconciliation. This activity is called “resolving.”

Often, the provider uses BDE- or ADO-enabled datasets such as you find in a two-tiered application. You can add database and session components as needed, just as in a BDE-based two-tiered application, or ADO connection components as in an ADO-based two-tiered application.

Note Do not confuse the ADO connection component, which is analogous to a database component in a BDE-based application, with the connection components used by client applications in a multitiered application.

For more information about two-tiered applications, see Chapter 13, “Building one- and two-tiered applications”.

If the application server is MTS-enabled, the MTS data module includes events for when the application server is activated or deactivated. This permits the application server to acquire database connections when activated and release them when deactivated.

Using MTS

Using MTS lets your remote data module take advantage of

- **MTS security.** 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 let you check the role of the currently connected client and conditionally allow certain functions based on that role. For more information about MTS security, see “Role-based security” on page 51-11.

- **Database handle pooling.** MTS data modules automatically pool database connections so that 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 component should set the *KeepConnection* property to *False*, so that your application maximizes the sharing of connections.
- **MTS transactions.** When using MTS, you can integrate your own MTS transactions into the application server to provide enhanced transaction support. MTS transactions can span multiple databases, or include functions that do not involve databases at all. For more information about MTS transactions, see “Managing transactions in multi-tiered applications” on page 14-25.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your MTS 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 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” on page 14-26.

By default, all automatically generated calls to an MTS data module are transactional (that is, they assume that when the call exits, the MTS data module can be deactivated and any current transactions can be committed or rolled back). You can write an MTS 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.

Warning When using MTS, database connections should not be opened until the remote 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, you must add code to open database connections when the data module is activated and to close them when the data module is deactivated.

Pooling remote data modules

Object pooling allows you some of the benefits available from MTS when you are not using DCOM. Under object pooling, you can 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 server 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.

Because a single instance of a remote data module potentially handles requests from several clients, it must not rely on persistent state information. See “Supporting state information in remote data modules” on page 14-26 for more information on how to ensure that your remote data module is stateless.

To take advantage of object pooling, your remote data module must override the *UpdateRegistry* method. In the overridden method, you can call *RegisterPooled* when the remote data module registers and *UnregisterPooled* when the remote data module unregisters.

You can only take advantage of object pooling when the connection is formed using HTTP.

Using the IAppServer interface

Remote data modules on the application server support the *IAppServer* interface. Connection components on client applications look for this interface to form connections.

IAppServer provides the bridge between client applications and the provider components in the application server. Most client applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of the client dataset.

Table 14.3 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 to indicate which provider on the application server should provide data or resolve updates. In addition, most methods include an *OleVariant* parameter called *OwnerData* that allows the client application and application server 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 application server to adjust for this information before and after each client call.

Table 14.3 AppServer interface members

IAppServer	Provider component	TClientDataSet
AS_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 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.

Using DCOM connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server. However, because DCOM is not included with Windows 95, client machines may not have DCOM installed.

DCOM provides the only approach that lets you use MTS security. MTS security is based on assigning roles to the callers of MTS objects. When calling into MTS using DCOM, DCOM informs MTS about the client application that generated the call. MTS can then 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. MTS can't assign roles to separate clients because, as far as MTS can tell, all calls to the application server are made by the runtime executable. For more information about MTS security, see "Role-based security" on page 51-11.

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 Chapter 30, "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 Chapter 29, "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*) which 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. For more information on object pooling, see “Pooling remote data modules” on page 14-7.

Unlike other connection components, you can't use callbacks when the connection is formed via HTTP.

Using OLEnterprise

OLEnterprise lets you use the Business Object Broker instead of relying on client-side brokering. The Business Object Broker provides load-balancing, fail-over, and location transparency.

When using OLEnterprise, you must install OLEnterprise runtime on both client and server systems. OLEnterprise runtime handles the marshalling of Automation calls and communicates between the client and server system using remote procedure calls (RPCs). For more information, see the OLEnterprise documentation.

Using CORBA connections

CORBA lets you integrate your multi-tiered database applications into an environment that is standardized on CORBA. For example, the MIDAS client for Java components rely on a CORBA connection. Because CORBA (and Java) is available on multiple platforms, this allows you to write cross-platform MIDAS applications. For more information about using CORBA in Delphi, see Chapter 28, “Writing CORBA applications”.

By using CORBA, your application automatically gets the benefits of load-balancing, location transparency, and fail-over from the ORB runtime software. In addition, you can add hooks to take advantage of other CORBA services.

Building a multi-tiered application

The general steps for creating a multi-tiered database application are

- 1 Create the application server.
- 2 Register or install the application server.
 - If the application server uses DCOM, HTTP, sockets, or OLEnterprise as a communication protocol, it acts as an Automation server and must be

registered like any other ActiveX or COM server. For information about registering an application, see “Registering an application as an Automation server” on page 47-5.

- If you are using MTS, the application server must be an Active Library rather than an .EXE. Because all COM calls must go through the MTS proxy, you do not register the application server. Instead, you install it with MTS. For information about installing libraries with MTS, see “Installing MTS objects into an MTS package” on page 51-21.
- When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic binding to your interface, you must install the server’s interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon). For more information about registering a CORBA server, see “Registering server interfaces” on page 28-8.

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 not using a Web connection or socket connection, you may want to register or install 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 or socket connection, the connection component fetches the names of registered servers 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 includes a dataset provider.

To create an application server, start a new project, save it, and follow these steps:

- 1 Add a new remote data module to the project. From the main menu, choose File | New. Choose the Multitier page in the new items dialog, and select
 - **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, sockets, or OLEEnterprise.
 - **MTS Data Module** if you are creating an Active Library that clients access using MTS. Connections can be formed using DCOM, HTTP, sockets, or OLEEnterprise.
 - **CORBA Data Module** if you are creating a CORBA server.

For more detailed information about setting up a remote data module, see “Setting up the remote data module” on page 14-13.

Note

Remote data modules are more than simple data modules. The CORBA remote data module acts as a CORBA server. Other data modules are COM Automation objects.

- 2 Place the appropriate dataset components on the data module and set them up to access the database server.
- 3 Place a *TDataSetProvider* component on the data module for each dataset. This provider is required for brokering client requests and packaging data.
- 4 Set the *DataSet* property for each provider component to the name of the dataset to access. There are additional properties that you can set for the provider. For more detailed information about setting up a provider, see Chapter 15, “Using provider components”.
- 5 Write application server code to implement events, shared business rules, shared data validation, and shared security. You may want to extend the application server’s interface to provide additional ways that the client application can call the server. For more information about extending the application server’s interface, see “Extending the application server’s interface” on page 14-16.
- 6 Save, compile, and register or install the application server.
 - When the application server uses DCOM, HTTP, sockets, or OLEnterprise as a communication protocol, it acts as an Automation server and must be registered like any other ActiveX or COM server. For information about registering an application, see “Registering an application as an Automation server” on page 47-5.
 - If you are using MTS, the application server must be an Active Library rather than an .EXE. Because all COM calls must go through the MTS proxy, you do not register the application server. Instead, you install it with MTS. For information about installing libraries with MTS, see “Installing MTS objects into an MTS package” on page 51-21.
 - When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic (late) binding to your interface, you must install the server’s interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon). For more information about registering CORBA servers, see “Registering server interfaces” on page 28-8.
- 7 If your server application does not use DCOM, 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.
 - For OLEnterprise, this is the OLEnterprise runtime.
 - For CORBA, this is the VisiBroker ORB.

Setting up the remote data module

When you set up and run an application server, it does not establish any connection with client applications. Instead, connection is maintained by client applications. The client application uses its connection component to establish a connection to the application server, which it uses to communicate with its selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

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” on page 14-4 for information on what type of remote data module you need.

Configuring TRemoteDataModule

To add a *TRemoteDataModule* component to your application, choose File | New and select Remote Data Module from the Multitier 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” on page 14-16.

If you are creating a DLL (Active Library), 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 are creating an EXE, you must 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 and select MTS Data Module from the Multitier page of the new items dialog. You will see the MTS 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” on page 14-16.

MTS applications are always DLLs (Active Libraries). You must specify the threading model in the MTS Data Module wizard. Choose Single, Apartment, Free, or Both.

- If you choose Single, MTS 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 or Free, you get the same thing: MTS ensures that any instance of your remote data module services one request at a time, but calls do not always use the same thread. You can’t use thread variables, because there is no guarantee that subsequent calls to the remote data module instance will 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. For more information on the shared property manager, see “Shared property manager” on page 51-12.
- If you choose Both, MTS calls into the remote data module’s interface in the same way as when you choose Apartment or Free. 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 and Free models under MTS are different than the corresponding models under DCOM.

You must also specify the MTS 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 an MTS 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 an MTS 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 MTS transactions.

Configuring TCorbaDataModule

To add a *TCorbaDataModule* component to your application, choose File | New and select CORBA Data Module from the Multitier page of the new items dialog. You will see the CORBA Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TCorbaDataModule* 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 *TCorbaDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note You can add your own properties and methods to your new interface. For more information on adding to your data module's interface, see "Extending the application server's interface" on page 14-16.

The CORBA Data Module wizard lets you specify how you want your server application to create instances of the remote data module. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the remote data module that handles all client requests. This is the model used in traditional CORBA development.
- When you choose instance-per-client, a new remote data module instance is created for each client connection. This instance persists until its timeout period elapses with no messages from the client. This allows the server to free instances when they are no longer used by clients, but holds the risk that the server may be freed prematurely if the client does not use the server's interface for a long time.

Note Unlike instancing for COM servers, where the model determines the number of instances of the process that run, with CORBA, instancing determines the number of instances created of your object. They are all created within a single instance of the server executable.

In addition to the instancing model, you must specify the threading model in the CORBA Data Module wizard. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each remote data module instance is guaranteed to receive only one client request at a time. You can safely access the objects contained in your remote data module. However, you must guard against thread conflicts when you use global variables or objects not contained in the remote data module.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared remote data module instance, because you must protect all use of objects contained in your remote data module.

Creating a data provider for the application server

Each remote data module on an application server contains one or more provider components. Each client dataset uses a specific provider, which acts as the bridge between the client dataset and the data it represents. A provider component (*TDataSetProvider*) takes care of packaging data into data packets that it sends to clients and applying updates received from the client.

Most of the data logic in the application server is handled by the provider components contained in the remote data module. Event handlers that respond to client requests implement your business and data logic, while properties on the provider component control what information is included in data packets. See Chapter 15, “Using provider components” for details on how to use a provider component to control the interaction with client applications.

Extending the application server’s interface

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. For more information about using the type library editor, see Chapter 50, “Working with type libraries”. Note that many of the features you can specify in the type library editor (such as help context, version, and so on) do not apply to CORBA interfaces. Any values you specify for these in the type library editor are ignored.

What Delphi does when you add new entries to the interface depends on whether you are creating a COM-based (*TRemoteDataModule* or *TMTSDDataModule*) or CORBA (*TCorbaDataModule*) server.

- When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).
- When you add to a CORBA interface, your changes are reflected in your unit source code and the automatically generated _TLB unit. The _TLB unit is added to the **uses** clause of your unit. You must add this unit to the **uses** clause in your client application if you want to take advantage of early binding. In addition, you can save an .IDL file from the type library editor using the Export to IDL button. The .IDL file is needed for registering the interface with the Interface Repository and Object Activation Daemon.

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.

Client applications call your interface extensions using the *AppServer* property of their connection component. For more information on how to do this, see “Calling server interfaces” on page 14-24.

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-based connection. *TWebConnection* does 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 the application server’s interface when using MTS

When using transactions or just-in-time activation under MTS, you must be sure all new methods call *SetComplete* to tell MTS when it is finished. This allows transactions to complete and so 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. This is because any communication that does not go through the remote data module’s

interface bypasses the MTS proxy, which can invalidate transactions. If you are using a stateless MTS data module, bypassing the MTS proxy can lead to crashes because you can't guarantee that the remote data module is active.

Creating the client application

In most regards, creating a multi-tiered client application is similar to creating a traditional two-tiered client. The major differences are that a multi-tiered client uses

- A connection component to establish a conduit to the application server.
- One or more *TClientDataSet* components to link to a data provider on the application server. Data-aware controls on the client are connected through data source components to these client datasets instead of *TTable*, *TQuery*, *TStoredProc* or *TADODataSet* components.

To create a multi-tiered client application, start a new project and follow these steps:

- 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" on page 14-4 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" on page 14-19.
- 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" on page 14-23
- 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 Chapter 24, "Creating and using a client dataset."
- 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 Create the client application in much the same way you would create any other database application. You will probably want to use some of the special features of client datasets that support their interaction with the provider components on the application server. These are described in "Using a client dataset with a data provider" on page 24-14.

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 MIDAS page of the Component 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” on page 14-8 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.
- Identify the application server on the server machine.

If you are not using CORBA, 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” on page 14-13 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.

If you are using CORBA, identify the server using the *RepositoryID* property. *RepositoryID* specifies the Repository ID of the application server’s factory interface, which appears as the third argument in the call to *TCorbaVCLComponentFactory.Create* that is automatically added to the initialization section of the CORBA server’s implementation unit. You can also set this property to the base name of the CORBA data module’s interface (the same string as the *ServerName* property for other connection components), and it is automatically converted into the appropriate Repository ID for you.

- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

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” on page 14-22.

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” on page 30-3.

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” on page 30-4.

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” on page 14-22.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher programs supplied with Delphi. 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 do this, create and register a COM object that supports the *IDataIntercept* interface. This is an interface

for encrypting and decrypting data. Next, set the *InterceptGUID* property of the socket connection component to the GUID for this COM object. Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept GUID to the same GUID. 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” on page 14-22.

Specifying a connection using OLEnterprise

When using OLEnterprise to communicate with the application server, client applications should include a *TOLEnterpriseConnection* component for connecting to the application server. When using OLEnterprise, you can either connect directly to the server machine, or you can use the Business Object Broker.

- To use OLEnterprise without going through a Business Object Broker, set the *ComputerName* property to the name of the server machine, just as you would use the *ComputerName* property for a DCOM connection.
- To use the load-balancing and fail-over services of the Business Object Broker, set the *BrokerName* property to the name of the Business Object Broker.

ComputerName and *BrokerName* are mutually exclusive. Setting the value of one unsets the value of the other.

For more information about using OLEnterprise, see the OLEnterprise documentation.

Specifying a connection using CORBA

Only the *RepositoryID* property is necessary in order to specify a CORBA connection. This is because a Smart Agent on the local network automatically locates an available server for your CORBA client.

However, you can limit the possible servers to which your client application connects by the other properties of the CORBA connection component. If you want to specify a particular server machine, rather than letting the CORBA Smart Agent locate any available server, use the *HostName* property. If there is more than one object instance that implements your server interface, you can specify which object you want to use by setting the *ObjectName* property.

The *TCorbaConnection* component obtains an interface to the CORBA data module on the application server in one of two ways:

- If you are using early (static) binding, you must add the *_TLB.pas* file (generated by the type library editor) to your client application. Early binding is highly recommended, both for compile-time type checking and because it is much faster than late (dynamic) binding.
- If you are using late (dynamic) binding, the interface must be registered with the Interface Repository. For more information about registering an interface with the Interface Repository, see “Writing CORBA clients” on page 28-11.

For more information on early vs. late binding, see “Calling server interfaces” on page 14-24.

Brokering connections

If you have multiple 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 Do not use the *ObjectBroker* property with OLEnterprise connections or CORBA connections. Both of these protocols have their own brokering services.

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.

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" on page 14-19. In addition, 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* 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* 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. You can do this using the *AppServer* property of the connection component. For more information about extending the application server's interface, see "Extending the application server's interface" on page 14-16.

AppServer is a Variant that represents the application server's interface. You can call an interface method using *AppServer* by writing a statement such as

```
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.

When you are using DCOM or CORBA 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:

```
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.

To use early binding with CORBA, you must add the `_TLB` unit that is generated by the type library editor to your project. To do this, add this unit to the **uses** clause of your unit.

When you are using TCP/IP or OLEnterprise, 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:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface := 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.

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 component or using passthrough SQL. 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 “Using transactions” on page 13-5 and “Working with (connection) transactions” on page 23-11.

If you are using MTS, you can broaden your transaction support by using MTS transactions. MTS 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, MTS transactions can span multiple databases.

Warning 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.

To use MTS transactions, extend the application server’s interface to include method calls that encapsulate the transaction, if necessary. When configuring the MTS remote data module indicate that it must participate in transactions. When a client calls a method on your application server’s 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 MTS transactions with explicit transactions created by a database component or using passthrough SQL. When your remote data module is enlisted in an MTS transaction, it automatically enlists all of your database calls in the transaction as well.

For more information about using MTS transactions, see “MTS transaction support” on page 51-7.

Supporting master/detail relationships

You can create master/detail relationships between client datasets in your client application in the same way you set up master/detail forms in one- and two-tiered applications. For more information about setting up master/detail forms, see “Creating master/detail forms” on page 20-24.

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” on page 24-16.
- 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 to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky. See “Applying updates for master/detail tables” on page 25-6 for more information on applying updates in traditional master/detail forms.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this, set up a master/detail relationship between the tables on the application server. Then set the *DataSet* property of your provider component to the master table.

When clients call the *GetRecords* method of the provider, it automatically includes the detail datasets 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.

See “Representing master/detail relationships” on page 24-3 for more information on using nested datasets to support master/detail relationships in client datasets.

Supporting state information in remote data modules

The *IAppServer* interface, which controls all communication between client datasets and 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 under MTS, 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 MTS just-in-time activation, object pooling, or typical CORBA servers.

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).

This is not a problem if the remote data module is configured so that each client has its own instance. When each client has its own instance of the remote data module, there are no other clients to change the state of the data module between client calls.

However, it is reasonable to want the benefits of sharing remote data module instances while still managing persistent state information. For example, you may need to use incremental fetching to display a dataset that is too large to fit in memory at one time.

Before and after any calls to the *IAppServer* interface that the client dataset sends to the application server (*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, to enable incremental fetching in a stateless application server, you can do the following:

- Use the client dataset's *BeforeGetRecords* event to send the key value of the last record to the application server:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
var
  CurRecord: TBookmark;
begin
  with Sender as TClientDataSet do
    begin
      CurRecord := GetBookmark; { save the current record }
      try
        Last; {locate the last record in the new packet }
        OwnerData := FieldValues['Key']; { Send key value to the application server }
        GotoBookmark(CurRecord); { return to current record }
      finally
        FreeBookmark(CurRecord);
      end;
    end;
  end;
```

- On the server, use the provider's *BeforeGetRecords* event to locate the appropriate set of records:

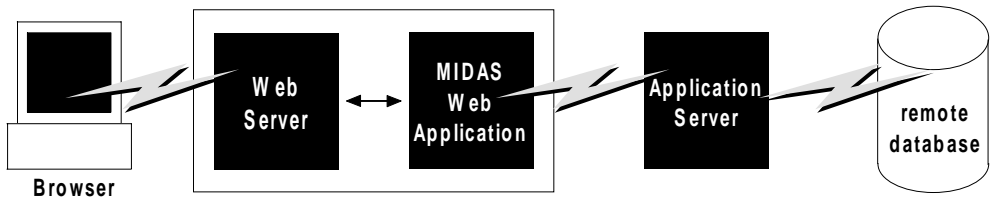
```
TRemoteDataModule1.Provider1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TProvider do
    DataSet.Locate('Key', OwnerData, []);
  end;
```

Note The previous example uses a key value to mark the end of the record set rather than a bookmark. This is because bookmarks may not be valid between *IAppServer* calls if the server application pools database handles.

Writing MIDAS Web 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 applications that acts simultaneously as a client to the application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in Figure 14.1.

Figure 14.1 Web-based multi-tiered database application



There are two approaches that you can take to build the MIDAS Web application:

- You can combine the MIDAS architecture with ActiveX support to distribute a 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.

Caution 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 MIDAS architecture can be combined with Delphi's 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. The only limitation is that you will want to use DCOM, HTTP, or sockets as a communications protocol, because you can't count on client machines having installed the OLEnterprise or CORBA runtime software. For details on creating the application server, see "Creating the application server" on page 14-11.

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", below, 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

- 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 Chapter 30, "Working with sockets."
- 2 Create the client application following the steps described in "Creating the client application" on page 14-18, except start by choosing File | New | Active Form, rather than beginning the client project as an ordinary Delphi 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:
 - 1 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 that is specified in the Windows Network | DNS settings.

- 2 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

MIDAS clients 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 Delphi's Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. These applications make up Delphi's InternetExpress support.

Before building an InternetExpress application, you should understand Delphi's Web server application architecture. This is described in Chapter 29, "Creating Internet server applications".

On the InternetExpress page of the component palette, you can find a set of components that extend this Web server application architecture to act as a MIDAS client. Using these components, the Web application generates 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.

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. See "Granting permission to access and launch the application server" on page 14-32 for details.

- Tip** You can use the components on the InternetExpress page to build Web server applications 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 how to build a Web application that creates HTML pages for allowing users to interact with the data from an application server via a javascript-enabled Web browser.

- 1 Choose File | New 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" on page 29-6.

- 2 From the MIDAS page of the component 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” on page 14-8 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” on page 14-19.
- 4 Instead of a client dataset, add an XML broker from the InternetExpress page of the component 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 its *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” on page 14-33.
- 6 Add a MIDAS page producer to the Web module for each separate page that users will see in their browsers. For each MIDAS 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” on page 29-9.
- 8 Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipses 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 a MIDAS page producer” on page 14-35.
- 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 scriptname 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 with Delphi:

Table 14.4 The javascript libraries

Library	functions
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.
xmlDB.js	This library defines data access classes analogous to TClientDataSet and TField.
xmlDisp.js	This library defines classes that associate the data access classes in xmlDB with HTML controls in the HTML page.

These libraries can be found in the Source/Webmidas directory. Once you have installed these libraries, you must set the *IncludePathURL* property of all MIDAS 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).

The classes in the javascript libraries are an evolving standard, and are updated regularly. If you want to use them directly rather than relying on Web items to generate the javascript code, you can get the latest versions and documentation of how to use them from CodeCentral at www.borland.com/CodeCentral.

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. The following steps describe how 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 of the application server, which it acquires through a connection component. You must set the following properties so that the XML producer can use the application server's *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:

- 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.
- You can limit the number of records that are added to the data packet by setting the *MaxRecords* 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" on page 29-9.

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 a MIDAS page producer

Each MIDAS 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 MIDAS page producer for each of them.

The MIDAS page producer 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" on page 29-12. For more information about page producers, see "Using page producer components" on page 29-17.

Unlike most page producers, the MIDAS page producer has a default template as the value of its *HTMLDoc* property. This template contains a set of HTML-transparent tags that the MIDAS 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 MIDAS page producer's the 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 MIDAS page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a MIDAS 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 MIDAS 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 (IE4).

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 also define a style sheet, that defines a set of style definitions. Each definition includes both a style selector (either 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 MIDAS page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.

- You can also define a style sheet, that includes a set of style definitions. Each definition consists of a style selector (either the name of a tag to which the style applies or a user-defined style name) and an 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 MIDAS 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 supply the style definitions as the value of the MIDAS 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 MIDAS page producer template

The template of a MIDAS page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the MIDAS 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 MIDAS 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 listed 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 MIDAS page producer automatically translates these tags when you call the *Content* method. In addition, The MIDAS 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 MIDAS 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 MIDAS 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 MIDAS 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" on page 29-18.

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 provider components

Provider components (*TDataSetProvider*) supply the mechanism by which client datasets obtain their data (unless they are using flat files). Providers are responsible for packaging data into data packets that are then sent to client datasets and applying updates received from client datasets. Usually they reside on an application server as part of a multi-tiered application, but they can appear as part of the same application as the client dataset (or XML broker). Providers work in conjunction with resolver components that handle the details of resolving data to the database or dataset.

Most of the work of a provider component happens automatically. You need not write any code on the provider to create a fully functioning application server. 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.

This chapter describes how to use a provider component to control the interaction with client applications.

Determining the source of data

When you use a provider component, you must specify a dataset that it can use to get the data it assembles into data packets. To do this, set the *DataSet* property of the provider to the name of the dataset to use. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

TDataSetProvider can work with any dataset that supports 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. Most of the dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, Client datasets, and InterBase Express components) override these methods to implement the *IProviderSupport* interface in a more useful fashion.

Note Because the provider relies on an interface belonging to the dataset, it has no specific dependencies on the data access mechanism (BDE, DBOLE, or some other mechanism). These dependencies all fall to the dataset that the provider uses.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to work in an application server. 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.

Choosing how to apply updates

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

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.

Specifying what fields appear in data packets

To control what fields are included in data packets, create persistent fields on the dataset that the provider uses to build the packets. The provider then includes only these fields. Fields whose values are generated dynamically on the server (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about creating persistent fields, see “Creating persistent fields” on page 19-5.

If the client dataset will be editing the data and applying updates to the application server, 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 using queries on the application server. The query should include enough fields so that records are unique, even if your application does not use all the fields.

Setting options that influence the data packets

The *Options* property of the provider component 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*.

Table 15.1 Provider options

Value	Meaning
poFetchBlobsOnDemand	BLOB field values are not included in the data packet. Instead, client applications must request these values on an as-needed basis. If the client dataset's <i>FetchOnDemand</i> property is <i>True</i> , the client requests these values automatically. Otherwise, the client application uses the client dataset's <i>FetchBlobs</i> method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider represents the master of a master/detail relationship, nested detail values are not included in the data packet. Instead, client applications request these on an as-needed basis. If the client dataset's <i>FetchOnDemand</i> property is <i>True</i> , the client requests these values automatically. Otherwise, the client application uses the client dataset's <i>FetchDetails</i> method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): <i>Alignment</i> , <i>DisplayLabel</i> , <i>DisplayWidth</i> , <i>Visible</i> , <i>DisplayFormat</i> , <i>EditFormat</i> , <i>MaxValue</i> , <i>MinValue</i> , <i>Currency</i> , <i>EditMask</i> , <i>DisplayValues</i> .
poCascadeDeletes	When the provider represents the master of a master/detail relationship, detail records are deleted by the server automatically 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 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.
poReadOnly	The client dataset can't apply updates to the provider.

Table 15.1 Provider options (continued)

Value	Meaning
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, custom SQL statements, 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.
poDisableEdits	Clients can't modify existing data values. If the client tries to edit a field an exception is raised. (This does not affect the client's ability to insert or delete records).
poDisableInserts	Clients can't insert new records. If the client tries to insert a new record an exception is raised. (This does not affect the client's ability to delete records or modify existing data)
poDisableDeletes	Clients can't delete new records. If the client tries to delete a record an exception is raised. (This does not affect the client's ability to insert or modify records)
poNoReset	Clients can't specify that the provider should reposition the cursor on the first record before providing data.
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poPropagateChanges	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.

Adding custom information to data packets

Providers can send application-defined information to the data packets using the *OnGetDataSetProperties* event. This information is encoded as an *OleVariant*, and stored under a name you specify. Client datasets in the client application 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 application may never be aware of the information, but the server 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. Multiple attributes can be added 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 information about the time the data was provided is returned when clients apply updates:

```
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;

```

When the client applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

```

procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TClientDataSet);
var
    WhenProvided: TDateTime;
begin
    WhenProvided := DataSet.GetOptionalParam('TimeProvided');
    ...
end;

```

Responding to client data requests

In most multi-tiered applications, client requests for data are handled automatically. A client dataset requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset, 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, the provider might want to encrypt sensitive data before it is sent on to the client, or to remove records from the packet based on some criterion (such as the user's role in an MTS application).

To edit the data packet before sending it on to the client, write an *OnGetData* event handler. The data packet is provided as a parameter in the form of a client dataset. Using the methods of this client dataset, data can be edited before it is sent to the client.

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client application before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 14-26.

Responding to client update requests

A provider applies updates to database records based on a *Delta* data packet received from a client application. The client requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client application before and after the call to *ApplyUpdates*. This communication takes place

using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers. For a discussion of persistent state information in application servers, see “Supporting state information in remote data modules” on page 14-26.

When a 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 *OnUpdateData* event, the provider uses its associated resolver component to write the changes to the database.

The resolver component performs the update on a record-by-record basis. Before the resolver applies each record, it generates a *BeforeUpdateRecord* event on the provider, which you can use to screen updates before they are applied. If an error occurs when updating a record, the resolver calls the provider’s *OnUpdateError* event handler to resolve the error. Usually errors occur because the change violates a server constraint or the database record was changed by a different application subsequent to its retrieval by this client application, but prior to the client’s request to apply updates.

Update errors can be processed by either the application server or the client. Application servers should handle all update errors that do not require user interaction to resolve. When the application server can’t resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the application server returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it passes back to the client 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 on the update status of records so that your event handler does not need to sort through records it won’t be using. To do this, set the *StatusFilter* property of the client dataset.

Note Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see “Applying updates to datasets that do not represent a single table” on page 15-9.

Editing delta packets before updating the database

Before the 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 Table 15.2.

Table 15.2 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:

```

procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TClientDataSet);
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;
    end;
end;

```

Influencing how updates are applied

The *OnUpdateData* event also gives your 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:

Table 15.3 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.
upWhereOnly	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 Table 15.4

Table 15.4 ProviderFlags values

Value	Description
pfnWhere	The field does not appear in the WHERE clause of generated INSERT, DELETE, and UPDATE statements.
pfnUpdate	The field does not appear in the UPDATE clause of generated UPDATE statements.
pfnKey	The field is used in the WHERE clause of a generated SELECT statement that executes when update failures occur. This SELECT statement tries to locate the current value of modified or deleted records, or a record causing key violations when insertions fail.
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 excludes the EMPNO field from the UPDATE clause and the TITLE and DEPT fields from the WHERE clause:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('EMPNO').UpdateFlags := [ufInUpdate];
    FieldByName('TITLE').UpdateFlags := [ufInWhere];
    FieldByName('DEPT').UpdateFlags := [ufInWhere];
  end;
end;
```

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, the 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 to the resolver that an update has been handled already and should not be applied. The resolver 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 application server tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the application server can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the application server returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

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 resolver component 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 *TTable* or "live" *TQuery* components.

Automatic updates are a problem however, if the resolver must apply updates to the data underlying a stored procedure with a result set or a multi-table query. This is

because 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 will use 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 resolver component can generate appropriate SQL statements to apply updates.

Note 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 resolver does not generate an error.

Responding to client-generated events

Provider components implement a general-purpose event that lets you create your own calls from clients 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 clients to communicate directly with providers on the application server. 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 have managed in the past. You can take advantage of server constraints in your multi-tiered database applications without having to duplicate the constraints in application server or client application code.

The the provider is working with a BDE-enabled dataset, its *Constraints* property enables you to replicate and apply server constraints to data passed to and received from client applications. When *Constraints* is *True* (the default), your server's constraints are replicated to clients and affect client attempts to update data.

Important Before the application server can pass constraint information on to the client application, it must retrieve the constraints from the database server. To import database constraints from the server, use SQL explorer to import the database server's constraints and default expressions into the Data Dictionary. Constraints and default expressions in the Data Dictionary are automatically made available to BDE-enabled datasets in the application server.

There may be times when you do not want to apply server constraints to data sent to a client application. For example, a client application 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 application server 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" on page 24-18.

Managing database sessions

Both standalone, full client database applications and database application servers can communicate with databases through the Borland Database Engine (BDE). An 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.

In an application, you can manage BDE sessions using the *TSession* and *TSessionList* components. Each *TSession* component in an application encapsulates a single BDE session. All sessions within an application are managed by a single *TSessionList* component.

All database applications automatically include a session component, named *Session*, that encapsulates the default BDE session. Applications can declare, create, and manipulate additional session components as needed.

All database applications also automatically include a session list component, named *Sessions*, that enables the application to manage all of its session components.

This chapter describes the session and session list components and explains how to use them to control BDE sessions in your full client database applications and database application servers.

Note The default session and session list components provide a widely applicable set of defaults that can be used as is by most applications. Only applications that use multiple sessions (for example, because of the need to run concurrent queries against a single database) may need to manipulate its session and session list components.

Working with a session component

A session component provides global management for a group of database connections within an application. When you create a full client database application or an application server, 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. Applications can control database connections and access to Paradox files by using the properties, events, and methods of the session.

You can use the default session to control all database connections in an 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.

Some applications require additional session components, such as applications that run concurrent queries against the same database. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions. Applications that use multiple sessions must manage those sessions through the session list component, *Sessions*. For more information about managing multiple sessions see, "Managing multiple sessions" on page 16-16.

Using the default session

All database applications automatically include a default session. Delphi creates a default session component named *Session* for a database application each time it runs (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are temporary (created by the session at runtime when a dataset is opened 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.

When you create a database component, it is automatically associated with the default session. 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. When creating a multi-threaded database application, you must create one additional session to handle each additional thread.

To use the default session, you need write no code unless your application must

- Modify the properties of the session, such as specifying when database connections for automatically generated database components should automatically be kept or dropped.
- Respond to session events, such as when the application attempts to access a password-protected Paradox file.
- Execute a session's methods, such as opening or closing a database in response to user-initiated actions.
- Set the *NetFileDir* property to access Paradox tables on a network and set the *PrivateDir* property to a local hard drive to speed performance.

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 your application opens 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. For more information about *KeepConnections*, see "Specifying default database connection behavior" on page 16-6. Additional properties, events, and methods of the *TSession* component which apply to the default session and any additional sessions you create in your applications are described in the rest of this chapter.

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. This section describes how to create and delete sessions at runtime.

Note Keep in mind that 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, follow these steps:

- 1 Declare a pointer to a *TSession* variable.
- 2 Instantiate a new session by calling the *Create* constructor. The constructor sets up an empty list of database components for the session, sets up an empty list of BDE callbacks 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" on page 16-4.
- 4 Activate the session and optionally adjust its properties.

Note Never delete the default session.

You can also manage creating and opening of 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 more information about using *OpenSession*, see "Managing multiple sessions" on page 16-16.

The following code creates a new session component, assigns it a name, and opens the session for database operations that follow (not shown here). After use, it is destroyed with a call to the *Free* method.

```

var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create;
  with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
  end;
end;

```

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.

For more information about using the *TSessionList* component—and *Sessions* in particular—to control multiple sessions, see "Managing multiple sessions" on page 16-16.

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:

```

var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;

```

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.

To determine the current state of a session, check *Active*. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

Setting *Active* to *True* triggers a session's *OnStartup* event, sets the *NetFileDir* and *PrivateDir* properties if they are assigned values, and sets the *ConfigMode* property. You can write an *OnStartup* event handler to perform other specific database start-up activities. For more information about *OnStartup*, see "Working with password-protected Paradox and dBase tables" on page 16-13. The *NetFileDir* and *PrivateDir* properties are only used for connecting to Paradox tables. For more information about them, see "Specifying the control file location" on page 16-13 and "Specifying a temporary files location" on page 16-13. *ConfigMode* determines how the BDE handles BDE aliases created within the context of the session. For more information about *ConfigMode*, see "Specifying alias visibility" on page 16-10. To open database components within a session, see "Creating, opening, and closing database connections" on page 16-6.

After you activate a session, 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 invoke 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.

For session components you create, use the Object Inspector to set *Active* to *False* at design time to disable all database access for a session with a single property change. You might want to do this if, during application design, you do not want to receive exceptions because a remote database is temporarily unavailable.

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:

```
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*:

```
Session1.Open;
```

Note If a session is active you can also open and close individual database connections. For more information, see "Closing a single database connection" on page 16-7.

Customizing session start-up

You can customize a session's start-up behavior by writing an *OnStartup* event handler for it. *OnStartup* is triggered when a session is activated. 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).

Specifying default database connection behavior

KeepConnections provides the default value for the *KeepConnection* property of temporary 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 "Creating, opening, and closing database connections" on page 16-6.

KeepConnections should always 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 inactive database connections for all temporary database components, and then free the temporary database components by calling the *DropConnections* method. For more information about *DropConnections*, see "Dropping temporary database connections" on page 16-8.

For example, the following code drops inactive connections and frees all temporary database components for the default session:

```
Session.DropConnections;
```

Creating, opening, and closing 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:

```
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

OpenDatabase makes its own session active if it is not already active, and then determines 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. 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. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server.

Closing a single database connection

You can close an individual database connection with the *CloseDatabase* method, or close all connections for a session at once with the *Close* method. When you call *CloseDatabase*, a reference count for the database is decremented by 1. When the reference count for the database is 0, the database is closed and freed. *CloseDatabase* takes one parameter, the database to close. For example, the following statement closes the database connection opened in the example in the previous section:

```
Session.CloseDatabase(DBDemosDatabase);
```

If the specified database name is associated with a temporary database component, and the session's *KeepConnections* property is *False*, the temporary 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.

If the database component is persistent (meaning that the application specifically declares the component and instantiates it), and the session's *KeepConnections* property is *False*, *CloseDatabase* calls the database component's *Close* method to close the connection.

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.

Closing all database connections

You can close all database connections in two ways:

- 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 temporary 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:

```
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 or not 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, the name 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:

```
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.Active then begin
    DB.StartTransaction;
    ...
  end;
end;
```

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. Table 16.1 summarizes the informational methods to a session component:

Table 16.1 Database-related informational methods for session components

Method	Purpose
<i>GetAliasDriverName</i>	Retrieves the BDE driver for a specified alias of a database.
<i>GetAliasNames</i>	Retrieves the list of BDE aliases for a database.
<i>GetAliasParams</i>	Retrieves the list of parameters for a specified BDE alias of a database.
<i>GetConfigParams</i>	Retrieves specific configuration information from the BDE configuration file.
<i>GetDatabaseNames</i>	Retrieves the list of BDE aliases and the names of any <i>TDatabase</i> components currently in use.
<i>GetDriverNames</i>	Retrieves the names of all currently installed BDE drivers.
<i>GetDriverParams</i>	Retrieves the list of parameters for a specified BDE driver.
<i>GetStoredProcNames</i>	Retrieves the names of all stored procedures for a specified database.
<i>GetTableNames</i>	Retrieves the names of all tables matching a specified pattern for 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:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

For a complete description of a session's informational methods, see the *TSession* topic in the VCL online help reference.

Working with BDE aliases

Because a session usually encapsulates a series of database connections, one property and many of a session component's methods work 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. BDE aliases and the associated *TSession* methods have three main uses:

- Determining visibility
- Retrieving alias and driver information
- Creating, modifying, and deleting aliases

The following sections describe these functional areas.

Specifying alias visibility

A session's *ConfigMode* property determines what BDE aliases are visible to the session. *ConfigMode* is a set that describes which types of sessions are visible. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session, all aliases in the BDE configuration file on a user's system, and all aliases that the BDE maintains in memory.

The main purpose of *ConfigMode* is to enable an application to specify and restrict alias visibility at the session level. 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.

For a full description of *ConfigMode* and its settings, see the online reference for the object and component library.

Making session aliases visible to other sessions and applications

When an alias is created during a session, the BDE stores a copy of the alias in memory. By default this copy is local only to the session in which it is created. Another session in the same application can only see the alias if its *ConfigMode* property is *cmAll* or *cfmPersistent*.

To make an 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.

Determining known aliases, drivers, and parameters

Five methods for session components enable an application to retrieve information about 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 a string containing 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 "Retrieving information about a session" on page 16-8. For more information about BDE aliases, parameters, and drivers, see the online BDE help file, BDE32.HLP.

Creating, modifying, and deleting aliases

A session can create, modify, and delete aliases during its lifetime. The *AddAlias* method creates a new BDE alias for an SQL database sever. *AddStandardAlias* creates a new BDE alias for Paradox, dBASE, or ASCII tables.

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 more information about *AddAlias*, see the topic for this method in the online VCL reference. For more information about BDE aliases and the SQL Links drivers, see the BDE online help, BDE32.HLP.

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 more information about *AddStandardAlias*, see the online reference for the object and component libraries. For more information about BDE aliases, see the BDE online help, BDE32.HLP.

Note When you add an alias to a session, it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. To make a newly created alias available to all applications, call *SaveConfigFile* after creating the alias. For more information about *ConfigMode*, see “Working with BDE aliases” on page 16-9.

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.

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.

The following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
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;
```

The following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

The following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

Iterating through a session's database components

Two session component properties, *Databases* and *DatabaseCount*, enable you to cycle through all the database components associated with a session.

Databases is an array of all currently active database components associated with a session. Used with the *DatabaseCount* property, *Databases* can be used to iterate through all active database components to perform a selective or universal action.

DatabaseCount is an Integer property that indicates the number of currently active databases associated with a session. As connections are opened or closed during a session's life-span, this number can 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.

DatabaseCount is typically used with the *Databases* property to perform actions common to active database components.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
  end;
```

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).

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements.

Specifying the control file location

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:

```
Session.NetFileDir := ExtractFilePath(Application.EXENAME);
```

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.

Specifying a temporary files location

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:

```
Session.PrivateDir := 'C:\TEMP';
```

Important Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

Working with password-protected Paradox and dBase tables

A session component provides four methods and one event that are exclusively used to manage access to password-protected Paradox and dBase files. The methods are *AddPassword*, *GetPassword*, *RemoveAllPasswords*, and *RemovePassword*. The event is

OnPassword. The *PasswordDialog* function is also available for adding and removing one or more passwords from a session.

Using the AddPassword method

The *TSession.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. *AddPassword* takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords to access files protected with different passwords.

```
var
  Passwrđ: String;
begin
  Passwrđ := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrđ);
  try
    Table1.Open
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

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. On a custom password entry form, use a *TEdit* with the *PasswordChar* set to an asterisk ("*").

The Add button of the *PasswordDialog* function dialog has the same effect as the *AddPassword* method.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

Note You need to call *AddPassword* to specify one or more passwords (one at a time) to use when accessing password-protected files. If you do not, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

Using the RemovePassword and RemoveAllPasswords methods

TSession.RemovePassword deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword('secret');
```

TSession.RemoveAllPasswords deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
```

Using the `GetPassword` method and `OnPassword` event

`TSession.GetPassword` triggers the `TSession.OnPassword` event for a session. The `OnPassword` event is called only when an application attempts to open a Paradox or dBase table for the first time and the BDE reports insufficient access rights. You can code an `OnPassword` event handler and provide a password to the BDE, or you can choose to use the default password handling, which displays a dialog box prompting the user for a password.

In the example below, the procedure `Password` is designated as the handler for the `OnPassword` event by assigning the procedure's name to the `TSession.OnPassword` property.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

In the `Password` procedure, the `InputBox` function is used to prompt the user for a password. The `TSession.AddPassword` method is used to programmatically supply the password entered in the dialog to the session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrld: String;
begin
  Passwrld := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrld > '');
  Session.AddPassword(Passwrld);
end;
```

The `Password` procedure is triggered by an attempt to open a password-protected table, as shown below. Exception handling can be used to accommodate a failed attempt to open the table. Even though the user is prompted for a password in the `OnPassword` event handler, the open attempt can still fail if they enter an invalid password or something else goes wrong.

```
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;
```

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 Data Access page on the Component palette contains a session component that you can place in a data module or on a form at design time.

Important 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 global function *Sessions.OpenSession* at runtime.

Sessions.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:

```
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 database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. Table 16.2 summarizes the properties and methods of the *TSessionList* component:

Table 16.2 TSessionList properties and methods

Property or Method	Purpose
<i>Count</i>	Returns the number of sessions, both active and inactive, in the sessions list.
<i>FindSession</i>	Searches the session list for a session with a specified name, and returns a pointer to the session component, or nil if there is no session with the specified name. If passed a blank session name, <i>FindSession</i> returns a pointer to the default session, <i>Session</i> .
<i>GetSessionNames</i>	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 (note that the default session's name is actually a blank string).
<i>List</i>	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
<i>OpenSession</i>	Creates and activates a new session or reactivates an existing session for a specified session name.
<i>Sessions</i>	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 a session component in data modules

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.

Connecting to databases

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 encapsulates the connection to a single database in the context of a BDE session. This chapter describes database components and how to manipulate database connections.

Database components are also used to manage transactions in BDE-based applications. For more information about using databases to manage transactions, see “Using transactions” on page 13-5.

Another use for database components is applying cached updates for related tables. For more information about using a database component to apply cached updates, see “Applying cached updates with a database component method” on page 25-5.

Understanding persistent and temporary database components

Each BDE-based database connection in an application is encapsulated by a database component whether or not you explicitly provide a database component at design time or create it dynamically at runtime. When an application attempts to connect to a database, it either uses an explicitly instantiated, or *persistent*, database component, or it generates a temporary database component that exists only for the lifetime of the connection.

Temporary database components are created as necessary for any datasets in a data module or form for which you do not create yourself. Temporary database components provide broad support for many typical desktop database applications without requiring you to handle the details of the database connection. For most client/server applications, however, you should create your own database components instead of relying on temporary ones. You gain greater control over your databases, including the ability to

- Create persistent database connections.

- Customize database server logins.
- Control transactions and specify transaction isolation levels.
- Create BDE aliases local to your application.

Using temporary database components

Temporary database components are automatically generated as needed. For example, if you place a *TTable* component on a form, set its properties, and open the table without first placing and setting up a *TDatabase* component and associating the table component with it, Delphi creates a temporary database component for you behind the scenes.

Some key properties of temporary database components are determined by the session to which they belong. 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 *OnPassword* event 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. Other properties of temporary database components provide standard login and transaction handling. For more information about sessions and session control over temporary database connections, see "Working with a session component" on page 16-1.

The default properties created for temporary database components provide reasonable, general behaviors meant to 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 database components to tune each database connection to your application's needs.

Creating database components at design time

The Data Access page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write *OnLogin* events for it. *OnLogin* offers you a chance to customize the handling of security on a database server when a database component first connects to the server. For more information about managing connection properties, see "Connecting to a database server" on page 17-6. For more information about server security, see "Controlling server login" on page 17-6.

Creating database components at runtime

You can create database components dynamically at runtime. An application might do this when the number of database components needed at runtime is unknown, and your application needs explicit control over the database connection. In fact, this is how Delphi itself creates temporary database components as needed at runtime.

When you create a database component at runtime, you need to give it a unique name and to associate it with a session.

You create the component by calling the *TDatabase.Create* constructor. Given both a database name and a session name, the following function creates a database component at runtime, associates it with a specified session (creating a new session if necessary), and sets a few key database component properties:

```
function TDataForm.RunTimeDbCreate(const DatabaseName, SessionName: String): TDatabase;
var
  TempDatabase: TDatabase;
begin
  TempDatabase := nil;
  try
    { If the session exists, make it active; if not, create a new session }
    Sessions.OpenSession(SessionName);
    with Sessions do
      with FindSession(SessionName) do begin
        Result := FindDatabase(DatabaseName);
        if (Result = nil) then begin
          { Create a new database component }
          TempDatabase := TDatabase.Create(Self);
          TempDatabase.DatabaseName := DatabaseName;
          TempDatabase.SessionName := SessionName;
          TempDatabase.KeepConnection := True;
        end;
        Result := OpenDatabase(DatabaseName);
      end;
    except
      TempDatabase.Free;
      raise;
    end;
  end;
end;
```

The following code fragment illustrates how you might call this function to create a database component for the default session at runtime:

```
var
  MyDatabase: array [1..10] of TDatabase;
  MyDbCount: Integer;
begin
  { Initialize MyDbCount early on }
  MyDbCount := 1;
  :
  { Later, create a database component at runtime }
  begin
    MyDatabase[MyDbCount] := RunTimeDbCreate('MyDb' + IntToStr(MyDbCount), '');
    Inc(MyDbCount);
  end;
  :
end;
```

Controlling connections

Whether you create a database component at design time or runtime, you can use the properties, events, and methods of *TDatabase* to control and change its behavior in your applications. The following sections describe how to manipulate database components. For details about all *TDatabase* properties, events, and methods, see the online reference for the object and component libraries.

Associating a database component with a session

All database components must be associated with a BDE session. Two database component properties, *Session* and *SessionName*, establish this association.

SessionName identifies the session alias with which to associate a database component. When you first create a database component at design time, *SessionName* is set to "Default". Multi-threaded or reentrant BDE applications may have more than one session. At design time, you can pick a valid *SessionName* from the drop-down list in the Object Inspector. Session names in the list come from the *SessionName* properties of each session component in the application.

The *Session* property is a runtime, read-only property that points to the session component specified by the *SessionName* property. For example, 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. This is useful when a database component is assigned to a different session at runtime.

For more information about BDE sessions, see Chapter 16, "Managing database sessions."

Specifying a BDE alias

AliasName and *DriverName* are mutually-exclusive BDE-specific properties. *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.

Note You create and edit BDE aliases using the Database Explorer or the BDE Administration utility. For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

DatabaseName enables you to provide an application-specific name for a database component. 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 enable you to link them to a database component.

DriverName is 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*.

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 enables you to view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. To work with connection parameters, see “Setting BDE alias parameters” on page 17-5. To set the state of *LoginPrompt*, see “Controlling server login” on page 17-6, and to set *KeepConnection* see “Connecting to a database server” on page 17-6.

To set *DatabaseName*, *AliasName*, or *DriverName* at runtime, include the appropriate assignment statement in your code. For example, the following code uses the text from an edit box to create a local alias for the database component *Database1*:

```
Database1.DatabaseName := Edit1.Text;
```

Setting BDE alias parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Administration utility 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. To learn more about the String List editor, see “Working with string lists” in the on-line help.
- 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. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links help file.

Controlling server login

Most remote database servers include security features to prohibit unauthorized access. Generally, 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:

- Set the *LoginPrompt* property of a database component to *True* (the default). Your application displays the standard Login dialog box when the server requests a user name and password.
- Set the *LoginPrompt* to *False*, and include hard-coded USER NAME and PASSWORD parameters in the *Params* property for the database component. For example:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

Important

Note that because the *Params* property is easy to view, this method compromises server security, so it is not recommended.

- Write an *OnLogin* event for the database component, and use it to set login parameters at runtime. *OnLogin* gets a copy of the database component's *Params* property, which you can modify. The name of the copy in *OnLogin* is *LoginParams*. Use the *Values* property to set or change login parameters as follows:

```
LoginParams.Values['USER NAME'] := UserName;
LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
```

On exit, *OnLogin* passes its *LoginParams* values back to *Params*, which is used to establish a connection.

Connecting to a database server

There are two ways to connect to a database server using a database component:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Setting *Connected* to *True* executes the *Open* method. *Open* verifies that the database specified by the *DatabaseName* or *Directory* properties exists, and if an *OnLogin* event exists for the database component, it is executed. Otherwise, the default Login dialog box appears.

Note When a database component is not connected to a server and an application attempts to open a dataset associated with the database component, the database component's *Open* method is first called to establish the connection. If the dataset is not associated with an existing database component, a temporary database component is created and used to establish the connection.

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, whether or not the connection is dropped depends on the setting of the database component's *KeepConnection* property.

KeepConnection determines if your application maintains a connection to a database even when all datasets associated with that database are closed. If *True*, a connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, make sure *KeepConnection* is *True* to reduce network traffic and speed up your application. If *False* a connection is dropped when there are no active datasets using the database. If a dataset is later opened which uses the database, the connection must be reestablished and initialized.

Special considerations when connecting to a remote server

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 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" on page 17-5.

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?
- 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?
- Are the DLLs for your connection and database drivers in the search path?

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.
- The BDE Administration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Administration utility. For more information, see the BDE Administration utility's online help file.

Disconnecting from a database server

There are two ways to disconnect a server from a database component:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Setting *Connected* to *False* calls *Close*. *Close* closes all open datasets and disconnects from the server. For example, the following code closes all active datasets for a database component and drops its connections:

```
Database1.Connected := False;
```

Note *Close* disconnects from a database server even if *KeepConnection* is *True*.

Closing datasets without disconnecting from a server

There may be times when you want to close all datasets without disconnecting from the database server. To close all open datasets without disconnecting from a server, follow these steps:

- 1 Set the database component's *KeepConnection* property to *True*.
- 2 Call the database component's *CloseDataSets* method.

Iterating through a database component's datasets

A database component provides two properties that enable an application to iterate through all the datasets associated with the component: *DataSets* and *DataSetCount*.

DataSets is an indexed array of all active datasets (*TTable*, *TQuery*, and *TStoredProc*) for a database component. An active dataset is one that is currently open. *DataSetCount* is a read-only integer value specifying the number of currently active datasets.

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 to set the *CachedUpdates* property for each dataset of type *TTable* to *True*:

```
var
  I: Integer;
begin
  for I := 0 to DataSetCount - 1 do
    if DataSets[I] is TTable then
      DataSets[I].CachedUpdates := True;
  end;
```

Understanding database and session component interactions

In general, session component properties, such as *KeepConnection*, provide global, default behaviors that apply to all temporary database components created as needed at runtime.

Session methods apply somewhat differently. *TSession* methods affect all database components, regardless of database component status. For example, the session method *DropConnections* closes 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.

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.

Executing SQL statements from a TDatabase component

Simple SQL statements can be executed directly from a *TDatabase* component using its *Execute* method. The *Execute* method is designed primarily for executing data definition language (DDL) SQL statements. These are statements that do not return

result sets and only operate on a database's metadata. The *Execute* method can, however, also be used to execute data manipulation language (DML) SQL statements that return result sets and operate on the data stored in the tables of a database.

Use the *Execute* method to execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, such as with SQL scripting utilities. To execute more than one statement: replace the contents of the *SQL* parameter with a new SQL statement and call the *Execute* method again, repeating these two steps as many times as there are statements to execute. It may be necessary or appropriate to make other changes as well, such as changing parameter values between statement executions, depending on the actual situation.

Executing SQL statements without result sets

Data definition language (DDL) SQL statements and some data manipulation language (DML) statements do not produce a result set. These statements are executed, affect metadata (for DDL statements) or data (DML statements), and then cease all interaction with the database. Examples of DDL statements include: CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Examples of DML statements that perform an action on data but do not return a result set include: INSERT, DELETE, and UPDATE.

To execute these statements that do not produce a result set, make a call to the *TDatabase.Execute* method. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the *SQL* parameter of the *Execute* method. If no parameters are used in the SQL statement, pass a **nil** value for the *Params* parameter of *Execute*. For information on using parameters with the *Execute* method, see "Executing parameterized SQL statements" on page 17-12. As these statements do not return a result set, pass a **nil** value in the *Cursor* parameter of *Execute*.

In the example below, a CREATE TABLE statement (DDL) without any parameters is executed with the *Execute* method.

```

procedure TDataForm.CreateTableButtonClick(Sender: TObject);
var
    SQLstmt: String;
begin
    Database1.Connected := True;
    SQLstmt := 'CREATE TABLE NewCusts ' +
        '( ' +
        '  CustNo INTEGER, ' +
        '  Company CHAR(40), ' +
        '  State CHAR(2), ' +
        '  PRIMARY KEY (CustNo) ' +
        ')';
    Database1.Execute(SQLstmt, nil, False, nil);
end;

```

In the following example, an INSERT statement (DDL) without any parameters and not returning a result set is executed.

```

procedure TDataForm.InsertRecordButtonClick(Sender: TObject);
var
    SQLstmt: String;
begin
    Database1.Connected := True;
    SQLstmt := 'INSERT INTO Customer ' +
        '(Custno, Company, State) ' +
        'VALUES (9999, "John Doe Rentals", "CA)';
    Database1.Execute(SQLstmt, nil, False, nil);
end;

```

Executing SQL statements with result sets

Only data manipulation language (DML) SQL statements return result sets. Even then, only a DML query that uses the SELECT statement produces a result set. (The SELECT statement might be against a base table or a stored procedure that returns a result set.)

To execute these statements that produce a result set, make a call to the *TDatabase.Execute* method and provide a dataset component for the result set. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the *SQL* parameter of the *Execute* method. If no parameters are used in the SQL statement, pass a **nil** value for the *Params* parameter of *Execute*. For information on using parameters with the *Execute* method, see “Executing parameterized SQL statements” on page 17-12.

To make the result set produced by the SQL statement available after the call to the *Execute* method, provide an already-existing dataset component (such as a *TTable* component). Provide a variable of type *hDBCICur* (a handle to a BDE cursor). (To make use of this data type, the unit containing the call to the *Execute* method must include the BDE wrapper unit “BDE” in its Uses section.) Pass this *hDBCICur* variable (dereferenced) in the *Cursor* parameter of the *Execute* method. After *Execute* is called, assign the handle of the BDE cursor to the *Handle* property of the dataset component. (The dataset component must be typecast as *TDBDataSet* to make this assignment because the *Handle* property is read-only in all classes related to datasets except *TDBDataSet*.)

The example routine below executes a SELECT statement using the *Execute* method. The result set produced by this action is made available through a *TTable* component named *Table1*.

```

procedure TDataForm.SELECT_NoParamsButtonClick(Sender: TObject);
var
    SQLstmt: String;
    Cursor: hDBCICur;
begin
    Database1.Connected := True;
    SQLstmt := 'SELECT Company, State ' +
        'FROM Customer ' +
        'ORDER BY State, Company';
    Database1.Execute(SQLstmt, nil, False, @Cursor);
    Table1.Close;
    (Table1 as TDBDataSet).Handle := Cursor;
end;

```

If the dataset component was used previously to represent a different SQL result set, call its *Close* method to properly deactivate that data cursor before assigning the new one to the dataset component.

The result set returned through the *Cursor* parameter of the *Execute* method is always read-only.

Executing parameterized SQL statements

Some SQL statements include parameters through which data values are passed.

To execute these statements that include parameters, make a call to the *TDatabase.Execute* method and pass an object of type *TParams* in the *Params* parameter. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the *SQL* parameter of the *Execute* method. If a statement does not return a result set, pass a **nil** value in the *Cursor* parameter. If the statement does return a result set, see “Executing SQL statements with result sets” on page 17-11.

To represent the parameters, use an object of type *TParams*. Pass this *TParams* object as the *Params* parameter of the *Execute* method. Each parameter of the SQL statement is represented in the *TParams* object by a separate *TParam* object. Use the *TParams.CreateParam* method to add one *TParam* object to the *TParams* object for each parameter in the SQL statement. Use properties and methods of *TParam* to set the value of the *TParam* object to give the SQL parameter a value when the *Execute* method is called.

In the example below, an SQL statement using parameters is executed. The SQL statement includes a single parameter named *StateParam*. A *TParams* object (called *stmtParams*) is created within the routine and the *TParams.CreateParam* method is called to add a single *TParam* object to *stmtParams*. After the *TParam* object has been created, it is assigned a value of “CA”. Then the *Execute* method is called using the *stmtParams* object for the *Params* parameter.

```

procedure TDataForm.SELECT_WithParamsButtonClick(Sender: TObject);
var
    SQLstmt: String;
    stmtParams: TParams;
    Cursor: hDBCur;
begin
    stmtParams := TParams.Create;
    try
        Database1.Connected := True;
        stmtParams.CreateParam(ftString, 'StateParam', ptInput);
        stmtParams.ParamByName('StateParam').AsString := 'CA';
        SQLstmt := 'SELECT Company, State ' +
            'FROM "Customer.db" ' +
            'WHERE (State = :StateParam) ' +
            'ORDER BY State, Company';
        Database1.Execute(SQLstmt, stmtParams, False, @Cursor);
        Table1.Close;
        TDBDataSet(Table1).Handle := Cursor;
    finally

```

```
    stmtParams.Free;  
end;  
end;
```

Successfully executing a parameterized SQL statement using the *Execute* method depends on these requirements being fulfilled:

- 1 Parameters must appear in the SQL statement (these being tokens prefixed with colons).
- 2 A *TParams* object must be created to contain *TParam* objects.
- 3 One *TParam* object must be created in the *TParams* object for each parameter in the SQL statement.
- 4 The *TParams* object must be used for the *Params* parameter of the *Execute* method.

If a parameter token is in the SQL statement but no *TParam* object is created to represent it, the parameter in the SQL statement cannot be given a value and the SQL statement may cause an error when executed (depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter token in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

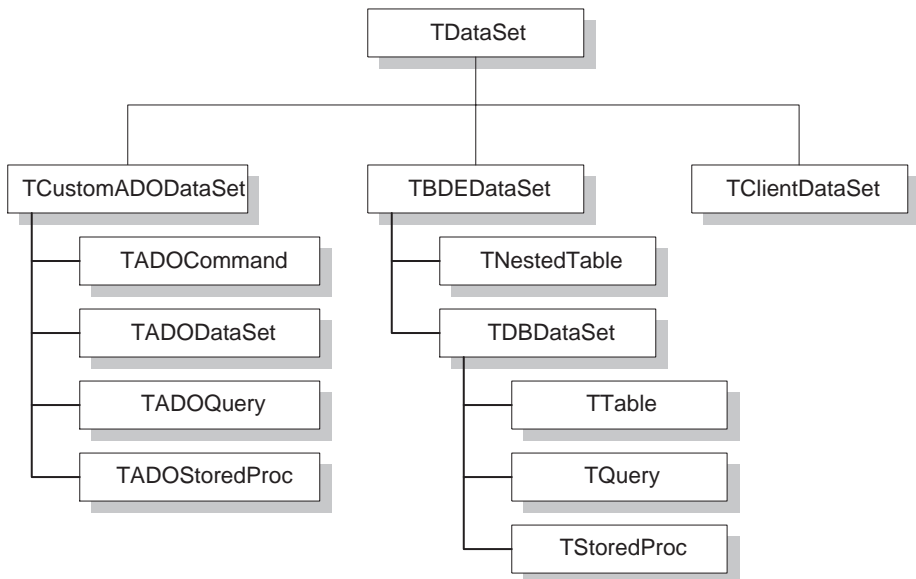
Understanding datasets

In Delphi, the fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. Generally, a dataset object represents a specific table belonging to a database, or it represents a query or stored procedure that accesses a database.

All dataset objects that you will use in your database applications descend from the virtualized dataset object, *TDataSet*, and they inherit data fields, properties, events, and methods from *TDataSet*. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you will use in your database applications. You need to understand this shared functionality to use any dataset object.

Figure 18.1 illustrates the hierarchical relationship of all the dataset components:

Figure 18.1 Delphi Dataset hierarchy



What is TDataSet?

TDataSet is the ancestor for all dataset objects you use in your applications. It defines a set of data fields, properties, events, and methods shared by all dataset objects. *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 *TDataSet*'s descendants, such as *TTable*, *TQuery*, *TStoredProc*, and *TClientDataSet*, 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.

Nevertheless, *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 more information about *TField* components, see Chapter 19, "Working with field components."

The following topics are discussed in this chapter:

- Types of datasets
- Opening and closing datasets
- Determining and setting dataset states
- Navigating datasets
- Searching datasets
- Displaying and editing a subset of data using filters
- Modifying data
- Using dataset events
- Using BDE-enabled datasets

Types of datasets

To understand the concepts common to all dataset objects, and to prepare for developing your own custom dataset objects that do not rely on either the Borland Database Engine (BDE) or ActiveX Data Objects (ADO), read this chapter.

To develop traditional, two-tier client/server database applications using the Borland Database Engine (BDE), see "Overview of BDE-enablement" discussed later in this chapter. That section introduces *TBDEDataSet* and *TDBDataSet*, and focuses

on the shared features of *TQuery*, *TStoredProc*, and *TTable*, the dataset components used most commonly in all database applications.

With some versions of Delphi, you can develop multi-tier database applications using distributed datasets. To learn about working with client datasets in multi-tiered applications, see Chapter 14, “Creating multi-tiered applications.” That chapter discusses how to use *TClientDataSet* and how to connect the client to an application server.

Opening and closing datasets

To read or write data in a table or through a query, an application must first open a dataset. You can open a dataset in two ways,

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

```
CustTable.Active := True;
```

- Call the *Open* method for the dataset at runtime,

```
CustQuery.Open;
```

You can close a dataset in two ways,

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime,

```
CustQuery.Active := False;
```

- Call the *Close* method for the dataset at runtime,

```
CustTable.Close;
```

You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TTable* component. At runtime, you may also want to close a dataset for other reasons specific to your application.

Determining and setting 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:

Table 18.1 Values for the dataset *State* property

Value	State	Meaning
<i>dsInactive</i>	Inactive	DataSet closed. Its data is unavailable.
<i>dsBrowse</i>	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.

Table 18.1 Values for the dataset State property (continued)

Value	State	Meaning
<i>dsEdit</i>	Edit	DataSet open. The current row can be modified.
<i>dsInsert</i>	Insert	DataSet open. A new row is inserted or appended.
<i>dsSetKey</i>	SetKey	<i>TTable</i> and <i>TClientDataSet</i> only. DataSet open. Enables setting of ranges and key values used for ranges and <i>GotoKey</i> operations.
<i>dsCalcFields</i>	CalcFields	DataSet open. Indicates that an <i>OnCalcFields</i> event is under way. Prevents changes to fields that are not calculated.
<i>dsCurValue</i>	CurValue	Internal use only.
<i>dsNewValue</i>	NewValue	Internal use only.
<i>dsOldValue</i>	OldValue	Internal use only.
<i>dsFilter</i>	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.

When an application opens a dataset, it appears by default in *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- code in your application, or
- built-in behavior of data-related components.

To put a dataset into *dsBrowse*, *dsEdit*, *dsInsert*, or *dsSetKey* states, call the method corresponding to the name of the state. For example, the following code puts *CustTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the database:

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if AddressPromptDialog.ModalResult := mrOK then
  CustTable.Post; { Delphi sets dataset state to Browse on successful completion }
else
  CustTable.Cancel; {Delphi sets dataset state to Browse on cancel }
```

This example also illustrates that the state of a dataset automatically changes to *dsBrowse* when

- The *Post* method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into *dsInactive* state, set its *Active* property to *False*, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable.Active := False;
CustTable.Close;
```

The remaining states (*dsCalcFields*, *dsCurValue*, *dsNewValue*, *dsOldValue*, and *dsFilter*) cannot be set by your application. Instead, the state of the dataset changes automatically to these values as necessary. For example, *dsCalcFields* is set when a

dataset's *OnCalcFields* event is called. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

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 "Using data sources" on page 26-5.

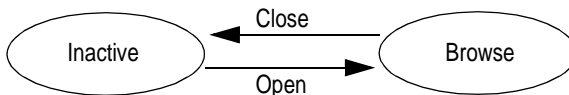
The following sections provide overviews of each state, how and when states are set, how states relate to one another, and where to go for related information, if applicable.

Inactivating a dataset

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time, a dataset is closed until you set its *Active* property to *True*. At runtime, a dataset is initially closed until an application opens it by calling the *Open* method, or by setting the *Active* property to *True*.

When you open an inactive dataset, its state automatically changes to the *dsBrowse* state. The following diagram illustrates the relationship between these states and the methods that set them.

Figure 18.2 Relationship of Inactive and Browse states



To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* modes when an application calls *Close*, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```

procedure CustTable.VerifyBeforeClose(DataSet: TDataSet)
begin
  if (CustTable.State = dsEdit) or (CustTable.State = dsInsert) then
    begin
      if MessageDlg('Post changes before closing?', mtConfirmation, mbYesNo, 0) = mrYes then
        CustTable.Post;
      else
        CustTable.Cancel;
      end;
    end;
end;

```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

- 1 Select the table in the data module (or form).
- 2 Click the Events page in the Object Inspector.
- 3 Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Browsing a dataset

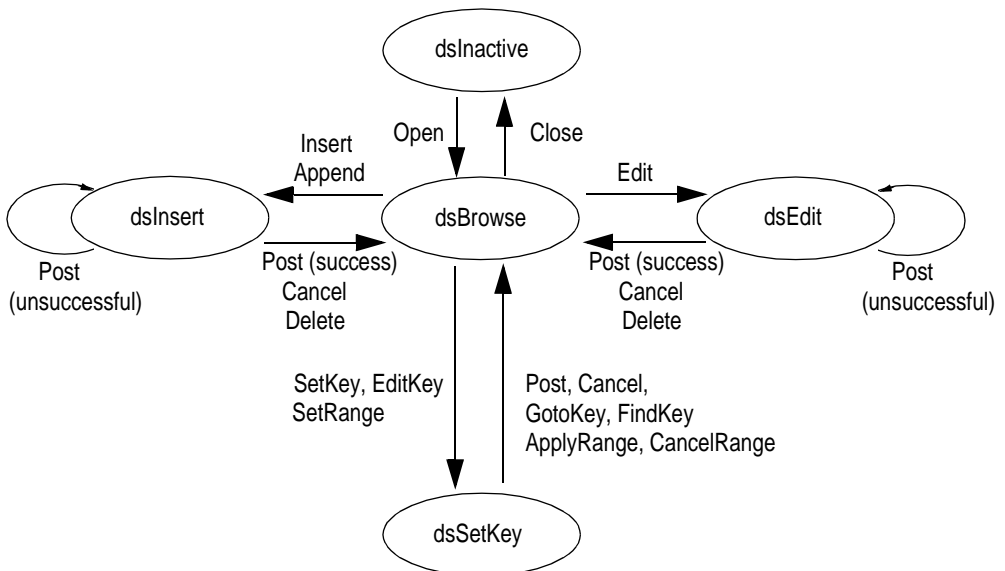
When an application opens a dataset, the dataset automatically enters *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset. For more information about scrolling from record to record, see “Navigating datasets” on page 18-9.

From *dsBrowse* all other dataset states can be set. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (note that other factors and dataset properties, such as *CanModify*, may prevent this change). Calling *SetKey* to search for records puts a dataset in *dsSetKey* mode. For more information about inserting and appending records in a dataset, see “Modifying data” on page 18-20.

Two methods associated with all datasets can return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

The following diagram illustrates the relationship of *dsBrowse* both to the other dataset modes you can set in your applications, and the methods that set those modes.

Figure 18.3 Relationship of Browse to other dataset states



Enabling dataset editing

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*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

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.

Important For *TTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records. Similarly, for *TQuery* components, if the *RequestLive* property is *False*, *CanModify* is *False*.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

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) or that causes the control to lose focus (such as moving to a different control on the form).

For a complete discussion of editing fields and records in a dataset, see “Modifying data” on page 18-20.

Enabling insertion of new records

A dataset must be in *dsInsert* mode before an application can add new records. In your 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*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

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),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important For *TTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records. Similarly, for *TQuery* components, if the *RequestLive* property is *False*, *CanModify* is *False*.

Note Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post*

attempts to write the new record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls for which inserting 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).

For more discussion of inserting and appending records in a dataset, see “Modifying data” on page 18-20.

Enabling index-based searches and ranges on tables

You can search against any dataset using the *Locate* and *Lookup* methods of *TDataSet*. *TTable* components, however, provide an additional family of *GotoKey* and *FindKey* methods that enable you to search for records based on an index for the table. To use these methods on table components, the component must be in *dsSetKey* mode. *dsSetKey* mode applies only to *TTable* components. You put a dataset into *dsSetKey* mode with the *SetKey* method at runtime. The *GotoKey*, *GotoNearest*, *FindKey*, and *FindNearest* methods, which carry out searches, returns the dataset to *dsBrowse* state upon completion of the search. For more information about searching a table based on its index, see “Searching for records based on indexed fields” on page 20-6.

You can temporarily view and edit a subset of data for any dataset by using filters. For more information about filters, see “Displaying and editing a subset of data using filters” on page 18-16. *TTable* components also support an additional way to access a subset of available records, called ranges. To create and apply a range to a table, a table must be in *dsSetKey* mode. For more information about using ranges, see “Working with a subset of data” on page 20-11.

Calculating fields

Delphi puts a dataset into *dsCalcFields* mode whenever an application calls the dataset’s *OnCalcFields* event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler is designed to modify. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see “Using OnCalcFields” on page 18-26.

Filtering records

Delphi puts a dataset into *dsFilter* mode whenever an application calls the dataset’s *OnFilterRecord* event handler. This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not

invalidated. For more information about filtering, see “Displaying and editing a subset of data using filters” on page 18-16.

When the *OnFilterRecord* handler finishes, the dataset is returned to *dsBrowse* state.

Updating records

When performing cached update operations, Delphi may put the dataset into *dsNewValue*, *dsOldValue*, or *dsCurValue* states temporarily. These states indicate that the corresponding properties of a field component (*NewValue*, *OldValue*, and *CurValue*, respectively) are being accessed, usually in an *OnUpdateError* event handler. Your applications cannot see or set these states.

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 values can be manipulated by edit, insert, and delete methods, and the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*.

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:

Table 18.2 Navigational methods of datasets

Method	Description
<i>First</i>	Moves the cursor to the first row in a dataset.
<i>Last</i>	Moves the cursor to the last row in a dataset.
<i>Next</i>	Moves the cursor to the next row in a dataset.
<i>Prior</i>	Moves the cursor to the previous row in a dataset.
<i>MoveBy</i>	Moves the cursor 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 more information about the navigator component, see Chapter 26, “Using data controls.”

In addition to these methods, the following table describes two Boolean properties of datasets that provide useful information when iterating through the records in a dataset.

Table 18.3 Navigational properties of datasets

Property	Description
<i>Bof</i> (Beginning-of-file)	<i>True</i> : the cursor is at the first row in the dataset. <i>false</i> : the cursor is not known to be at the first row in the dataset.
<i>Eof</i> (End-of-file)	<i>True</i> : the cursor is at the last row in the dataset. <i>false</i> : the cursor is not known to be at the last row in the dataset.

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*:

```
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*:

```
CustTable.Last;
```

Note While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you should enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that when active and visible enables 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 Chapter 26, “Using data controls.”

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*:

```
CustTable.Next;
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* 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*:

```
CustTable.Prior;
```

Using the MoveBy method

MoveBy enables you to 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.

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*:

```
CustTable.MoveBy(-2);
```

Note If you use *MoveBy* in your application and you work 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 because several users may be 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 for controlling dataset navigation, particularly 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 terminates when *Eof* is *True*. Inside the loop, call *Next* for each record in the dataset. *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*:

```
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;

```

Tip This example also demonstrates 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 Delphi does not have 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*:

```

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* and its descendants implement a bookmarking feature that enables you to tag records

and return to them later. The bookmarking feature consists of a *Bookmark* property and five bookmark methods.

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.

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. Descendants of *TDataSet*, such as *TBDEDataSet*, reimplement the bookmark methods to return meaningful values as described in the following list:

- *BookmarkValid*, for determining if a specified bookmark is in use.
- *CompareBookmarks*, to test two bookmarks to see if they are the same.
- *GetBookmark*, to allocate a bookmark for your current position in the dataset.
- *GotoBookmark*, to return to a bookmark previously created by *GetBookmark*
- *FreeBookmark*, to free a bookmark previously allocated by *GetBookmark*.

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* variable is a pointer (void *).

Before calling *GotoBookmark* to move to a specific record, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record. In *TDataSet*, *BookmarkValid* is a virtual method that always returns *False*, indicating that the bookmark is not valid. *TDataSet* descendants reimplement this method to provide a meaningful return value.

You can also call *CompareBookmarks* to see if a bookmark you want to move to is different from another (or the current) bookmark. *TDataSet.CompareBookmarks* always returns 0, indicating that the bookmarks are identical. *TDataSet* descendants reimplement this method to provide a meaningful return value.

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. *TDataSet.GotoBookmark* calls an internal pure virtual method which generates a runtime error if called. *TDataSet* descendants reimplement this method to provide a meaningful return value.

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.

The following code illustrates one use of bookmarking:

```

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;

```

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

You can search any dataset for specific records using the generic search methods *Locate* and *Lookup*. These methods enable you to search on any type of columns in any dataset.

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. 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.”:

```

var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
    SearchOptions);
end;

```

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 enables you to 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 type as an argument (for example, the return values from the *Lookup* method), or you must construct the variant array on the fly using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
    Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

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:

```
var
    LookupResults: Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
            Contact; Phone');
    end;
```

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. For more information about variant arrays, see the online help.

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 semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the variant array on the fly using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
var
    LookupResults: Variant;
begin
    with CustTable do
```

```
LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),  
    'Company; Addr1; Addr2; State; Zip');  
end;
```

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 within 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.

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 whether or not 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.

Enabling and disabling filtering

Enabling filters on a dataset is a three-step process:

- 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:

- Specify simple filter conditions in 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 conditions. The string 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:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

You can also supply a value for *Filter* based on the text entered in a control. For example, the following statement assigns the text in an edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

You can, of course, create a string based on both hard-coded text and data entered by a user in a control:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to *True*.

You can also compare field values to literals, and to constants using the following comparison and logical operators:

Table 18.4 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

Table 18.4 Comparison and logical operators that can appear in a filter (continued)

Operator	Meaning
AND	Tests two statements are both <i>True</i>
NOT	Tests that the following statement is not <i>True</i>
OR	Tests that at least one of two statements is <i>True</i>

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

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is 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 filter handler must set an *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”:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
    Accept := DataSet['State'] = 'CA';
end;
```

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 of your application.

Switching filter event handlers at runtime

You can code any number of filter event handlers and switch among them at runtime. To switch to a different filter event handler at runtime, assign the new event handler to the dataset’s *OnFilterRecord* property.

For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```


Setting filter options

The *FilterOptions* property enables you to specify whether or not a filter that compares string-based fields accepts records based on partial comparisons and whether or not 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:

Table 18.5 FilterOptions values

Value	Meaning
<i>foCaseInsensitive</i>	Ignore case when comparing strings.
<i>foPartialCompare</i>	Disable partial string matching (i.e., do not match strings ending with an asterisk (*)).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := ''State' = 'CA'';
```

Navigating records in a filtered dataset

There are four dataset methods that enable you to navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Table 18.6 Filtered dataset navigational methods

Method	Purpose
<i>FindFirst</i>	Move to the first record in the dataset that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
<i>FindLast</i>	Move to the last record in the dataset that matches the current filter criteria.
<i>FindNext</i>	Moves from the current record in the filtered dataset to the next one.
<i>FindPrior</i>	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:

```
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 whether or not filtering is currently enabled for the dataset. 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:

Table 18.7 Dataset methods for inserting, updating, and deleting data

Method	Description
<i>Edit</i>	Puts the dataset into <i>dsEdit</i> state if it is not already in <i>dsEdit</i> or <i>dsInsert</i> states.
<i>Append</i>	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in <i>dsInsert</i> state.
<i>Insert</i>	Posts any pending data, and puts the dataset in <i>dsInsert</i> state.
<i>Post</i>	Attempts to post the new or altered record to the database. If successful, the dataset is put in <i>dsBrowse</i> state; if unsuccessful, the dataset remains in its current state.
<i>Cancel</i>	Cancels the current operation and puts the dataset in <i>dsBrowse</i> state.
<i>Delete</i>	Deletes the current record and puts the dataset in <i>dsBrowse</i> state.

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*. *CanModify* is *True* if the table(s) underlying a dataset permits read and write privileges.

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.

Important For *TTable* components with the *ReadOnly* property set to *True* and *TQuery* components with the *RequestLive* property set to *False*, *CanModify* is *False*, preventing editing of records.

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 provide a navigator component on your forms, users can cancel edits by clicking the navigator's Cancel button. Cancelling 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,

```
with CustTable do
begin
    Edit;
    FieldValues['CustNo'] := 1234;
    Post;
end;
```

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 to the database.

Note If the *CachedUpdates* property for a dataset is *True*, posted modifications are written to a temporary buffer. To write cached edits to the database, call the *ApplyUpdates* method for the dataset. For more information about cached updates, see Chapter 25, "Working with cached updates."

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*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

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.

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 cached updates are enabled, *Post* writes the record to a buffer. To write cached inserts and appends to the database, call the *ApplyUpdates* method for the dataset.

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 cached updating is enabled), 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 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 cached updating is enabled), 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 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

A dataset must be active before an application can delete records. *Delete* deletes the current record from a dataset and puts the dataset in *dsBrowse* mode. The record that followed the deleted record becomes the current record. If cached updates are enabled for a dataset, a deleted record is only removed from the temporary cache buffer 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 to the database

The *Post* method is central to a Delphi application's interaction with a database table. *Post* writes changes to the current record to the database, but it behaves differently depending on a dataset's state.

- In *dsEdit* state, *Post* writes a modified record to the database (or buffer if cached updates is enabled).
- In *dsInsert* state, *Post* writes a new record to the database (or buffer if cached updates is enabled).
- In *dsSetKey* state, *Post* returns the dataset to *dsBrowse* state.

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.

Note 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.

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:

Table 18.8 Methods that work with entire records

Method	Description
<i>AppendRecord</i> ([array of values])	Appends a record with the specified column values at the end of a table; analogous to <i>Append</i> . Performs an implicit <i>Post</i> .
<i>InsertRecord</i> ([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to <i>Insert</i> . Performs an implicit <i>Post</i> .
<i>SetFields</i> ([array of values])	Sets the values of the corresponding fields; analogous to assigning values to <i>TFields</i> . Application must perform an explicit <i>Post</i> .

These method take an array of *Tavern* values as an argument, where each value corresponds to a column in the underlying dataset. Use the `ARRAYOFCONST` macro to create these arrays. 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 tables, both methods places 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:

```
CountryTable.InsertRecord(['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:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
```

```

    Post;
end;
end;

```

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.

Warning When using NULL pointers with *SetFields* to leave some field values untouched, be sure to cast the NULL to a void *. If you use NULL as a parameter without the cast, you will set the field to a blank value.

Using dataset events

Datasets have a number of events that enable an application to perform validation, compute totals, and perform other tasks. The events are listed in the following table.

Table 18.9 Dataset events

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

For more information about events for the *TDataSet* component, see the online *VCL Reference*.

Aborting a method

To abort a method such as an *Open* or *Insert*, call the *Abort* procedure in any of the *Before* event handlers (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

```

procedure TForm1.TableBeforeDelete (Dataset: TDataSet)begin
    if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
        Abort;
end;

```

Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, then *OnCalcFields* is called when

- A dataset is opened.
- 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.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of *AutoCalcFields*.

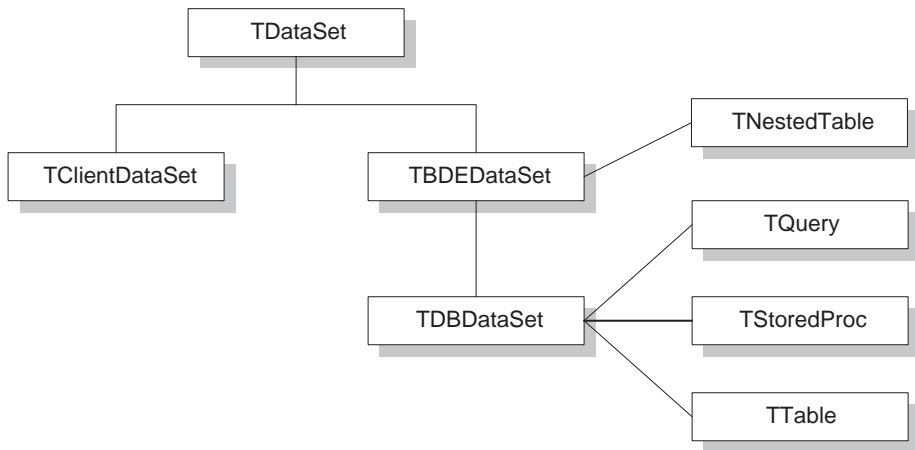
Caution *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, leading to another *Post*, and so on.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a single record are modified.

When *OnCalcFields* executes, a dataset is in *dsCalcFields* mode, so you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Using BDE-enabled datasets

BDE-enabled datasets provide functionality to the dataset components that use the Borland Database Engine (BDE) to access data. Support for BDE-enablement occurs in *TBDEDataSet*, which is a direct descendant of *TDataSet*. Additional database and session control features occur in *TDBDataSet*, which is a direct descendant of *TBDEDataSet*.

Figure 18.4 Dataset component hierarchy

This section introduces the dataset features provided by *TBDEDataSet* and *TDBDataSet*. It assumes you are already familiar with *TDataSet* discussed earlier in this chapter. For a general understanding of dataset components descended from *TDataSet*, see the beginning of this chapter.

Note Although you need to understand the functionality provided by *TBDEDataSet* and *TDBDataSet*, unless you develop your own custom BDE-enabled datasets, you never use *TBDEDataSet* and *TDBDataSet* directly in your applications. Instead, you use the direct descendants of *TDBDataSet*: *TQuery*, *TStoredProc*, and *TTable*. For specific information about using *TStoredProc*, see Chapter 22, “Working with stored procedures.” For specific information about using *TQuery*, see Chapter 21, “Working with queries.” For specific information about *TTable*, see Chapter 20, “Working with tables.”

Overview of BDE-enablement

The *TBDEDataSet* component implements the abstract methods of *TDataSet* that control record navigation, indexing, and bookmarking. It also reimplements many of *TDataSet*'s virtual methods and events to take advantage of the BDE. The BDE-specific implementations of *TDataSet*'s features do not depart from the general description about using these features with *TDataSet*, so for more information about them, see at the beginning of this chapter.

In addition to BDE-specific features common to all datasets, *TBDEDataSet* introduces new properties, events, and methods for handling BLOBs, cached updates, and communicating with a remote database server. *TDBDataSet* introduces a method and properties for handling database connections and associating a dataset with a BDE session. The following sections describe these features and point to other sections in the *Developer's Guide* that are also relevant to using them.

Handling database and session connections

The *TDBDataSet* component introduces the following properties and function for working with database and session connections:

Table 18.10 TDBDataSet database and session properties and function

Function or property	Purpose
<i>CheckOpen</i> function	Determines if a database is open. Returns <i>True</i> if the connection is active, <i>False</i> otherwise.
<i>Database</i>	Identifies the database component with which the dataset is associated.
<i>DBHandle</i>	Specifies the BDE database handle for the database component specified in the <i>Database</i> property. Used only when making some direct BDE API calls.
<i>DBLocale</i>	Specifies the BDE locale information for the database component specified in the <i>Database</i> property. Used only when making some direct BDE API calls.
<i>DBSession</i>	Specifies the BDE session handle for the session component specified by the <i>SessionName</i> property. Used only when making some direct BDE API calls.
<i>DatabaseName</i>	Specifies the BDE alias or database component name for the database used by this dataset. If the dataset is a Paradox or dBASE table, <i>DatabaseName</i> can be a full path specification for the database's directory location.
<i>SessionName</i>	Specifies the session with which this dataset component is associated. If you use both database and session components with a dataset, the setting for <i>SessionName</i> should be the same as the database component's <i>SessionName</i> property.

Using the DatabaseName and SessionName properties

Of the *TDBDataSet* database and session properties, the most commonly used are *DatabaseName* and *SessionName*. If you work with databases on a remote database server, such as Sybase, Oracle, or InterBase, your application usually maintains that connection through a *TDatabase* component. You should set the *DatabaseName* property of each dataset to match the name of the database component that establishes the database connection used by the dataset. If you do not use database components, *DatabaseName* should be set to a BDE alias (or, optionally, a full path specification for dBASE and Paradox).

SessionName indicates the BDE session with which to associate a dataset. If you do not use explicit session components in your application, you do not have to provide a value for this property. It is supplied for you. If your application provides more than one session, you can set a dataset's *SessionName* property to match the *SessionName* property of the appropriate session component in your application. If your application uses both multiple session components and one or more database components, the *SessionName* property for a dataset must match the *SessionName* property for the database component with which the dataset is associated.

For more information about handling database connections with *TDatabase*, see Chapter 17, "Connecting to databases." For more information about managing sessions with *TSession* and *TSessionList*, see Chapter 16, "Managing database sessions."

Working with BDE handle properties

Unless you bypass the built-in functionality of dataset components and make direct API calls to the BDE, you do not need to use the *DBHandle*, *DBLocale*, and *DBSession* properties. These properties are read-only properties that are automatically assigned to a dataset when it is connected to a database server through the BDE. These properties are provided as a resource for application developers who need to make direct API calls to BDE functions, some of which take handle parameters. For more information about the BDE API, see the online help file, BDE32.HLP.

Using cached updates

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache 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.

The recommended approach when caching updates is to use a client dataset rather than a BDE-enabled dataset. However, *TBDEDataSet* provides an alternate approach, with built-in methods for handling cached updates. The following table lists the relevant properties, events, and methods for cached updating:

Table 18.11 Properties, events, and methods for cached updates

Property, event, or method	Purpose
<i>CachedUpdates</i> property	Determines whether or not cached updates are in effect for the dataset. If <i>True</i> , cached updating is enabled. If <i>False</i> , cached updating is disabled.
<i>UpdateObject</i> property	Indicates the name of the <i>TUpdateSQL</i> component used to update datasets based on queries.
<i>UpdatesPending</i> property	Indicates whether or not the local cache contains updated records that need to be applied to the database. <i>True</i> indicates there are records to update. <i>False</i> indicates the cache is empty.
<i>UpdateRecordTypes</i> property	Indicates the kind of updated records to make visible to the application during the application of cached updates.
<i>UpdateStatus</i> method	Indicates if a record is unchanged, modified, inserted, or deleted.
<i>OnUpdateError</i> event	A developer-created procedure that handles update errors on a record-by-record basis.
<i>OnUpdateRecord</i> event	A developer-created procedure that processes updates on a record-by-record basis.
<i>ApplyUpdates</i> method	Applies records in the local cache to the database.
<i>CancelUpdates</i> method	Removes all pending updates from the local cache without applying them to the database.
<i>CommitUpdates</i> method	Clears the update cache following successful application of updates.
<i>FetchAll</i> method	Copies all database records to the local cache for editing and updating.
<i>RevertRecord</i> method	Undoes updates to the current record if updates are not yet applied on the server side.

Using cached updates and coordinating them with other applications that access data in a multi-user environment is an advanced topic that is fully covered in Chapter 25, “Working with cached updates.”

For information about using a client dataset instead, see Chapter 24, “Creating and using a client dataset”.

Caching BLOBs

TBDEDataSet provides the *CacheBlobs* property to control 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 field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Descendants of *TField* represent individual database columns in datasets. This chapter also describes how to use descendant field components to control the display and editing of data in your applications.

You never use a *TField* component directly in your applications. By default, when you first place a dataset in your application and open it, Delphi automatically assigns a dynamic, data-type-specific descendant of *TField* to represent each column in the database table(s). At design time, you can override dynamic field defaults by invoking the Fields editor to create persistent fields that replace these defaults.

The following table lists each descendant field component, its standard purpose, and, where appropriate, the range of values it can represent:

Table 19.1 Field components

Component name	Purpose
<i>TADTField</i>	An ADT (Abstract Data Type) field.
<i>TAggregateField</i>	A maintained aggregate in a client dataset.
<i>TArrayField</i>	An array field.
<i>TAutoIncField</i>	Whole number with a range of -2,147,483,648 to 2,147,483,647. Used in Paradox for fields whose values are automatically incremented.
<i>TBCDField</i>	Real number with a fixed number of decimal places, accurate to 18 digits. Range depends on the number of decimal places.
<i>TBooleanField</i>	<i>True</i> or <i>False</i> values.
<i>TBlobField</i>	Binary data: Theoretical maximum limit: 2GB.
<i>TBytesField</i>	Binary data: Theoretical maximum limit: 2GB.
<i>TCurrencyField</i>	Real numbers with a range of $5.0 * 10^{-324}$ to $1.7 * 10^{308}$. Used in Paradox for fields with two decimals of precision.
<i>TDataSetField</i>	Nested data set value.
<i>TDateField</i>	Date value.

Table 19.1 Field components (continued)

Component name	Purpose
<i>TDateTimeField</i>	Date and time value.
<i>TFloatField</i>	Real numbers with a range of $5.0 * 10^{-324}$ to $1.7 * 10^{308}$.
<i>TBytesField</i>	Binary data: Maximum number of bytes: 255.
<i>TIntegerField</i>	Whole number with a range of -2,147,483,648 to 2,147,483,647.
<i>TLargeIntField</i>	Whole number with a range of -2^{63} to 2^{63} .
<i>TMemoField</i>	Text data: Theoretical maximum limit: 2GB.
<i>TNumericField</i>	Real numbers with a range of $3.4 * 10^{-4932}$ to $1.1 * 10^{4932}$.
<i>TReferenceField</i>	A pointer to an object relational database object.
<i>TSmallIntField</i>	Whole number with a range of -32,768 to 32,768.
<i>TStringField</i>	String data: Maximum size in bytes: 8192, including a null termination character.
<i>TTimeField</i>	Time value.
<i>TVarBytesField</i>	Binary data: Maximum number of bytes: 255.
<i>TWordField</i>	Whole numbers with a range of 0 to 65,535.

This chapter discusses the properties and methods all field components inherit from *TField*. In many cases, *TField* declares or implements standard functionality that the descendant objects override. When several descendant objects share overridden functionality, that functionality is also described in this chapter and noted for your convenience. For complete information about individual field components, see the online *VCL Reference*.

Understanding field components

Like all Delphi data access components, field components are nonvisual. Field components are also not directly visible at design time. Instead they are associated with a dataset component and provide data-aware components such as *TDBEdit* and *TDBGrid* access to database columns through that dataset.

Generally speaking, a single field component represents the characteristics of a single column in a database field, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. Finally, as you scroll from record to record within a dataset, a field component also enables you to view and change the value for that field in the current record. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

Table 19.2 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.

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 sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

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, associate the dataset with a database, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the columns in one or more database tables accessed by a dataset. Delphi generates one field component for each column in the underlying tables or query. The exact *TField* descendant created for each column in an underlying database table is determined by field type information received from the Borland Database Engine (BDE) or (in multi-tiered applications) a provider component.

A field component's type determines its properties and how data associated with that field is displayed in data-aware controls on a form. Dynamic fields are temporary. They exist only as long as a dataset is open.

Each time you reopen a dataset that uses dynamic fields, Delphi rebuilds a completely new set of dynamic field components for it based on the current structure of the database tables underlying the dataset. If the columns in those database tables are changed, 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 exploration tool like the SQL Explorer, 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 database tables 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 database tables and queries, and associate the data sources with the datasets.
- 3 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.

4 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 so that you can set or change the field's visibility or display characteristics at design time or runtime, create new fields based on existing fields in a dataset, or validate data entry, you must create persistent fields for the 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.

Creating persistent field components offers the following advantages. You can:

- Restrict the fields in your dataset to a subset of the columns available in the underlying database.
- Add field components to the list of persistent components.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields—usually to replace existing fields—based on columns in the table or query underlying a dataset.
- Define calculated fields that compute their values based on other fields in the dataset.
- Define lookup fields that compute their values based on fields in other datasets.
- Modify field component display and edit properties.

A persistent field is one that Delphi generates based on field names and properties you specify in the Fields editor. 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 choose Add Fields from 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” on page 19-3.

Note One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control data appearance in other ways. For example, you can use the Data Dictionary to assign field attributes to a field component. You can also control the appearance of columns in data-aware grids. For more information about the Data Dictionary, see “Creating attribute sets for field components” on page 19-14. To learn about controlling column appearance in grids, see “Creating a customized grid” on page 26-17.

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 Set the *DatabaseName* property for the dataset.
- 3 Set the *TableName* property (for a *TTable*), or the *SQL* property (for a *TQuery*).
- 4 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 enable you to 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.

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.

- 5 Choose Add Fields from the Fields editor context menu.
- 6 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 Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, Delphi 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, Delphi verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, it 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. The following table lists the types of additional fields you can create:

Table 19.3 Special persistent field kinds

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type, based on columns in the table or query underlying a dataset.)
Calculated	Displays values calculated at runtime by a dataset's <i>OnCalcFields</i> event handler.
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify.
Aggregate	Displays a summary value of the data in a set of records.

These types of persistent fields are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and 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 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 type radio group enables you to 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're working with a *TClientDataSet* component, you can also create InternalCalc fields.

The Field properties group box enables you to enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. Delphi 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* contains the identifier by which you refer to the field component in your source code). Delphi discards anything you enter directly in the Component edit box.

The Type combo box in the Field properties group enables you to 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. The Size edit box enables you 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 Lookup definition group box is only used to create lookup fields. For more information, see "Defining a lookup field" on page 19-9.

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.

Important

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, follow these steps:

- 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" on page 19-12.

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 in the Field type radio group.
- 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 **type** declaration in the source code.
- 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" on page 19-9.

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" on page 19-12.

If you are working with a client dataset or query component, you can also create an `InternalCalc` field. You create and program an internally calculated field just like you do a calculated field. For 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. To create an `InternalCalc` field, select the `InternalCalc` radio button in the Field type group.

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:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
    + ' ' + CustomersZip.Value;
```

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.

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 this behavior. *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.

- Tip** You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Create a *TLookupList* object at runtime, and use its *Add* method to fill it with lookup values. Set the *LookupList* property of the lookup field to this *TLookupList* object and set its *LookupCache* property to *True*. If the other lookup properties of the field are not set, the field will use the supplied lookup list without overwriting it with values from a lookup dataset.

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 *tRefreshLookupList* 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.

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's *Aggregates* is automatically updated to include the appropriate aggregate specification, and the component declaration is automatically added to the form's **type** declaration in the source code.
- 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" on page 24-10.

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 re-create persistent field components that you

delete by accident, but any changes previously made to its properties or events is lost. For more information, see “Creating persistent fields” on page 19-5.

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 *TDBG*Grid, 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.

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.

Table 19.4 Field component properties

Property	Purpose
<i>Alignment</i>	Left justifies, right justifies, or centers a field contents within a data-aware component.
<i>ConstraintErrorMessage</i>	Specifies the text to display when edits clash with a constraint condition.
<i>CustomConstraint</i>	Specifies a local constraint to apply to data during editing.
<i>Currency</i>	Numeric fields only. <i>True</i> : displays monetary values. <i>False</i> (default): does not display monetary values.
<i>DisplayFormat</i>	Specifies the format of data displayed in a data-aware component.
<i>DisplayLabel</i>	Specifies the column name for a field in a data-aware grid component.
<i>DisplayWidth</i>	Specifies the width, in characters, of a grid column that display this field.
<i>EditFormat</i>	Specifies the edit format of data in a data-aware component.
<i>EditMask</i>	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).
<i>FieldKind</i>	Specifies the type of field to create.
<i>FieldName</i>	Specifies the actual name of a column in the table from which the field derives its value and data type.
<i>HasConstraints</i>	Indicates whether or not there are constraint conditions imposed on a field.
<i>ImportedConstraint</i>	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.

Table 19.4 Field component properties (continued)

Property	Purpose
<i>Index</i>	Specifies the order of the field in a dataset.
<i>LookupDataSet</i>	Specifies the table used to look up field values when <i>Lookup</i> is <i>True</i> .
<i>LookupKeyFields</i>	Specifies the field(s) in the lookup dataset to match when doing a lookup.
<i>LookupResultField</i>	Specifies the field in the lookup dataset from which to copy values into this field.
<i>MaxValue</i>	Numeric fields only. Specifies the maximum value a user can enter for the field.
<i>MinValue</i>	Numeric fields only. Specifies the minimum value a user can enter for the field.
<i>Name</i>	Specifies the component name of the field component within Delphi.
<i>Origin</i>	Specifies the name of the field as it appears in the underlying database.
<i>Precision</i>	Numeric fields only. Specifies the number of significant digits.
<i>ReadOnly</i>	<i>True</i> : Displays field values in data-aware components, but prevents editing. <i>False</i> (the default): Permits display and editing of field values.
<i>Size</i>	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of <i>TBytesField</i> and <i>TVarBytesField</i> fields.
<i>Tag</i>	Long integer bucket available for programmer use in every component as needed.
<i>Transliterate</i>	<i>True</i> (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. <i>False</i> : Locale translation does not occur.
<i>Visible</i>	<i>True</i> (the default): Permits display of field in a data-aware grid component. <i>False</i> : 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” on page 19-15.

Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
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.

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.

Note You can also create attribute sets directly from the SQL Explorer. When you create an attribute set from the data dictionary, it is not applied to any fields, but you can specify two additional attributes: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCkCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged to the form. For more information, see the online help for the SQL Explorer.

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.

Important 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 to multi-select field components within a dataset to which to reapply attributes.

Removing attribute associations

If you change your mind about associating an attribute set with a field, you can remove the association by following these steps:

- 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.

Important 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* 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 enables you to 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*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Table 19.5 Field component formatting routines

Routine	Used by ...
<i>FormatFloat</i>	<i>TFloatField</i> , <i>TCurrencyField</i>
<i>FormatDateTime</i>	<i>TDateField</i> , <i>TTimeField</i> , <i>TDateTimeField</i>
<i>FormatCurr</i>	<i>TCurrencyField</i>

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. For more information about setting field component properties at runtime, see “Setting field component properties at runtime” on page 19-13.

Handling events

Like most components, field components have event handlers associated with them. By writing these handlers you can control events that affect data entered in fields through data-aware controls. The following table lists the events associated with field components:

Table 19.6 Field component events

Event	Purpose
<i>OnChange</i>	Called when the value for a field changes.
<i>OnGetText</i>	Called when the value for a field component is retrieved for display or editing.
<i>OnSetText</i>	Called when the value for a field component is set.
<i>OnValidate</i>	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:

```
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 the entries for *TField* and its descendants in the online *VCL Reference*.

Table 19.7 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 Chapter 26, “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.

Displaying field component values in standard controls

An application can access the value of a database column through the *Value* property of a field component. For example, the following statement assigns the value of the *CustomersCompany* field to the text in a *TEdit* control:

```
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 functions for handling conversions.

Note You can also use variants to access and set field values. Variants are a new and flexible data type. For more information about using variants to access and set field values, see “Accessing field values with the default dataset property” on page 19-20.

Converting field values

Conversion functions attempt to convert one data type to another. For example, the *AsString* function converts numeric and Boolean values to string representations. The following table lists field component conversion functions, and which functions are recommended for field components by field-component type:

Table 19.8 Field component conversion functions

Function	TStringField	TIntegerField	TSmallIntField	TWordField	TFloatField	TCurrencyField	TBCDField	TDateTimeField	TDateField	TTimeField	TBooleanField	TBytesField	TVarBytesField	TBlobField	TMemoField	TGraphicField
<i>AsVariant</i>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
<i>AsString</i>		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
<i>AsInteger</i>	✓				✓	✓	✓									
<i>AsFloat</i>	✓	✓	✓	✓				✓	✓	✓						
<i>AsCurrency</i>	✓	✓	✓	✓				✓	✓	✓						
<i>AsDateTime</i>	✓															
<i>AsBoolean</i>	✓															

Note that the *AsVariant* method is recommended to translate among all data types. 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. Table 19.9 lists permissible conversions that produce special results:

Table 19.9 Special conversion results

Conversion	Result
<i>String to Boolean</i>	Converts “True,” “False,” “Yes,” and “No” to Boolean. Other values raise exceptions.
<i>Float to Integer</i>	Rounds float value to nearest integer value.
<i>DateTime to Float</i>	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
<i>Boolean to String</i>	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.

You use a conversion function as you would use any method belonging to a component: append the function name to the end of the component name wherever it occurs in an assignment statement. Conversion always occurs before an actual 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:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

An exception occurs if an unsupported conversion is performed at runtime.

Accessing field values with the default dataset property

The preferred 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:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

For more information about variants, see the online help.

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. Accessing field values with a dataset's *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 the field component's conversion routine. For more information about field component conversion functions, see "Converting field values" on page 19-18.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.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:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accessing field values with a dataset's FieldByName method

You can also 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

function, 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:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Checking a field's current value

If your application uses *TClientDataSet* or administers a dataset that is the source dataset for a *TProvider* component on an application server, and you encounter difficulties when updating records, you can use the *CurValue* property to examine the field value in the record causing problems. *CurValue* represents the current value of the field component including changes made by other users of the database.

Use *CurValue* to examine the value of a field when a problem occurs in posting a value to the database. If the current field value is causing a problem, such as a key violation, when posting the value to the database, an *OnReconcileError* occurs. In an *OnReconcileError* event handler, *NewValue* is the unposted value that caused the problem, *OldValue* is the value that was originally assigned to the field before any edits were made, and *CurValue* is the value that is currently assigned to the field. *CurValue* may differ from *OldValue* if another user changed the value of the field after *OldValue* was read.

Setting a default value for a field

You can specify how a default value for a field 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.

Working with constraints

Field components can use SQL server constraints. In addition, your applications can create and use custom constraints 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. The following sections describe working with constraints at the field component level.

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 pre-validation 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.

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, Delphi offers 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 field (*TObjectField*) descendants support the field types ADT (Abstract Data Type), Array, DataSet, and Reference. All of these field types either contain or reference child fields or other data sets.

ADT fields and reference fields map to fields that contain child fields. An ADT field contains child fields, which themselves can be any scalar or object type. An array field contains an array of child fields, all of the same type.

Dataset and reference fields map to fields that access other data sets. A dataset field provides access to a nested data set and a reference field stores a pointer (reference) to another persistent object (ADT).

Table 19.10 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 fields to be stored hierarchically rather than flattened out.

The following properties are common to all object field descendants and provide the functionality to handle child fields and datasets.

Table 19.11 Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classification of the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields of the object field.

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* and *TDBGrid* automatically display ADT and array field types.

Data-aware controls with a *DataField* property automatically displays any ADT and array fields and their child fields in the drop-down list. When an ADT or array field is bound to a data-aware control, the child fields appear in an uneditable comma

delimited string in the control. A child field is bound to the control as a normal data field.

A *TDBGGrid* 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.

Working with ADT fields

ADTs are user-defined types created on the server, and are similar to structures. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

Accessing ADT field values

There are a variety of ways to access the data in ADT field types. Creating and using persistent fields is strongly recommended. The following examples assign a child field value to an edit box called *CityEdit*, and uses the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

and the following persistent fields created for the *Customer* table component,

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

This line of code uses a persistent field and demonstrates the recommended method of accessing data in ADT fields.

```
CityEdit.Text := CustomerAddrCity.AsString;
```

The following code examples require that the dataset's *ObjectView* property be set to *True* in order to compile. They don't require persistent fields.

This example uses a fully qualified name with the *FieldByName* method on the dataset.

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

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 takes an integer value which specifies the offset of the

field. It is also the default property on *TObjectField*, and can therefore be omitted. For example,

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

which is the same as,

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

This code uses the *Fields* property of the *TADTField* component.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

This code uses the *Fields* property of the *TADTField* component with *FieldByName* of both the dataset and the *TFields* object.

```
CityEdit.Text :=
  TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

As you can see from this last example, accessing the field's data through persistent fields is much simpler. These additional access methods are primarily useful when the structure of the database table is not fixed or known at design time.

ADT field values can also be accessed with a dataset's *FieldValues* property:

```
Customer.Edit;
Customer['Address.City'] := CityEdit.Text;
Customer.Post;
```

The next statement reads a string value from the *City* child field of the ADT field *Address* into an edit box:

```
CityEdit.Text := Customer['Address.City'];
```

Note The dataset's *ObjectView* property can be either *True* or *False* for these lines of code to compile.

Working with array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (e.g. 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.

Accessing array field values

There are a variety of ways to access the data in array field types. The following example populates a list box with all of the non-null array elements.

```
var
  OrderDates: TArrayField;
  I: Integer;
begin
  for I := 0 to OrderDates.Size - 1 do
  begin
    if OrderDates.Fields[I].IsNull then Break;
    OrderDateListBox.Items.Add(OrderDates[I]);
  end;
end;
```

The following examples assign a child field value to an edit box called *TelEdit*, and uses the array *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component are used by the following examples:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

This line of code uses a persistent field to assign an array element value to an edit box.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

The following code examples require that the dataset's *ObjectView* property be set to *True* in order to compile. They don't require persistent fields.

You can access the value of a child field with the dataset's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

which is the same as,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

This next code example uses the *Fields* property of the *TArrayField* component.

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).Fields[1].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 a "(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.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

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. This particular *TDataSet* descendant is *TNestedTable*, and it provides the specific functionality needed to access data stored in nested datasets. Once a *TDataSetField* is associated with a dataset field, persistent fields can be created for the fields of the nested dataset.

To access the data in a dataset field you first create a persistent *TDataSetField* object by invoking the table's Fields editor, and then link to this field using the *DataSetField* property on a *TNestedTable* or *TClientDataSet* object. If the nested dataset field for the current record is assigned, the nested dataset will contain records with the nested data; otherwise, the nested 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.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a reference field

To access the data in a reference field you first create a persistent *TDataSetField*, and then link to this field using the *DataSetField* property on a *TNestedTable* or *TClientDataSet*. If the reference is assigned, the reference will contain a single record with the referenced data. If the reference is null, the reference will be empty.

The following examples are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.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:

```
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;
```


Working with tables

This chapter describes how to use the *TTable* dataset component in your database applications. A table component encapsulates the full structure of and data in an underlying database table. A table component inherits many of its fundamental properties and methods from *TDataSet*, *TBDEDataSet*, and *TDBDataSet*. Therefore, you should be familiar with the general discussion of datasets in “Understanding datasets,” and the BDE-specific discussion of datasets in “Using BDE-enabled datasets” on page 18-26 before reading about the unique properties and methods of table components discussed here.

Using table components

A table component gives you access to every row and column in an underlying database table, whether it is from Paradox, dBASE, Access, FoxPro, an ODBC-compliant database, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

You can view and edit data in every column and row of a table. You can work with a range of rows in a table, and you can filter records to retrieve a subset of all records in a table based on filter criteria you specify. You can search for records, copy, rename, or delete entire tables, and create master/detail relationships between tables.

Note A table component always references a single database table. If you need to access multiple tables with a single component, or if you are only interested in a subset of rows and columns in one or more tables, you should use a query component instead of a table component. For more information about query components, see Chapter 21, “Working with queries.”

Setting up a table component

The following steps are general instructions for setting up a table component at design time. There may be additional steps you need to tailor a table's properties to the requirements of your application.

- To create a table component,
 - 1 Place a table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
 - 2 Set the *DatabaseName* of the component to the name of the database to access.
 - 3 Set the *TableName* property to the name of the table in the database. You can select tables from the drop-down list if the *DatabaseName* property is already specified.
 - 4 Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.
- To access the data encapsulated by a table component,
 - 1 Place a data source component from the Data Access page of the Component palette in the data module or form, and set its *DataSet* property to the name of the table component.
 - 2 Place a data-aware control, such as *TDBGrid*, on a form, and set the control's *DataSource* property to the name of the data source component placed in the previous step.
 - 3 Set the *Active* property of the table component to *True*.

Specifying a database location

The *DatabaseName* property specifies where the table component looks for a database table. For Paradox and dBASE, *DatabaseName* can be a Borland Database Engine (BDE) alias, or an explicit directory path. For SQL tables, *DatabaseName* must be a BDE alias.

The advantage of using BDE aliases in all cases is that you can change the data source for an entire application by simply changing the alias definition in the SQL Explorer. To change the alias definition using SQL explorer, right click the SQL explorer and select Rename. This displays the BDE Administration Tool. For more information about setting and using BDE aliases, see the online help for the SQL Explorer.

To set the *DatabaseName* property,

- 1 Set the table's *Active* property to *False* if necessary.
- 2 Specify the BDE alias or directory path in the *DatabaseName* property.

Tip If your application uses database components to control database transactions, *DatabaseName* can be set to a local alias defined for the database component instead. For more information about database components, see Chapter 17, “Connecting to databases.”

Specifying a table name

The *TableName* property specifies the table in a database to access with the table component. To specify a table, follow these steps:

- 1 Set the table's *Active* property to *False*, if necessary.
- 2 Set the *DatabaseName* property to a BDE alias or directory path. For more information about setting *DatabaseName*, see “Specifying a database location” on page 20-2.
- 3 Set the *TableName* property to the table to access. At design time you can choose from valid table names in the drop-down list for the *TableName* property in the Object Inspector. At runtime, you must specify a valid name in code.

Once you specify a valid table name, you can set the table component's *Active* property to *True* to connect to the database, open the table, and display and edit data.

At runtime, you can set or change the table associated with a table component by:

- Setting *Active* to *False*.
- Assigning a valid table name to the *TableName* property.

For example, the following code changes the table name for the *OrderOrCustTable* table component based on its current table name:

```
with OrderOrCustTable do
begin
  Active := False; {Close the table}
  if TableName = 'CUSTOMER.DB' then
    TableName := 'ORDERS.DB'
  else
    TableName := 'CUSTOMER.DB';
  Active := True; {Reopen with a new table}
end;
```

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 an application accesses SQL-based tables on database servers.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its file-name extension. Table 20.1 summarizes the file

name extensions recognized by the BDE and the assumptions it makes about a table's type:

Table 20.1 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 Table 20.1, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. Table 20.2 indicates the values you can assign to *TableType*:

Table 20.2 TableType values

Value	Table type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

Opening and closing a table

To view and edit a table's data in a data-aware control such as *TDBGrid*, open the table. There are two ways to open a table. You can set its *Active* property to *True*, or you can call its *Open* method. Opening a table puts it into *dsBrowse* state and displays data in any active controls associated with the table's data source.

To end display and editing of data, or to change the values for a table component's fundamental properties (e.g., *DatabaseName*, *TableName*, and *TableType*), first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database. Finally, close the table.

There are two ways to close a table. You can set its *Active* property to *False*, or you can call its *Close* method. Closing a table puts the table into *dsInactive* state. Active controls associated with the table's data source are cleared.

Controlling read/write access to a table

By default when a table 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).

There are three properties for table components that can affect an application's read and write access to a table: *CanModify*, *ReadOnly*, and *Exclusive*.

CanModify is a read-only property that specifies whether or not a table component is permitted read/write access to the underlying database table. After you open a table at runtime, your application can examine *CanModify* to test whether or not the table has write access. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided that the table's *ReadOnly* property is *False*.

ReadOnly determines whether or not 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 a table.

Exclusive controls whether or not an 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:

```
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 may 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.

Searching for records

You can search for specific records in a table in various ways. The most flexible and preferred way to search for a record is to use the generic search methods *Locate* and *Lookup*. These methods enable you to search on any type of fields in any table, whether or not they are indexed or keyed.

- *Locate* finds the first row matching a specified set of criteria and moves the cursor to that row.
- *Lookup* returns values from the first row that matches a specified set of criteria, but does not move the cursor to that row.

You can use *Locate* and *Lookup* with any kind of dataset, not just *TTable*. For a complete discussion of *Locate* and *Lookup*, see Chapter 18, "Understanding datasets."

Table components also support the *Goto* and *Find* methods. While these methods are documented here to allow you to work with legacy applications, you should always use *Lookup* and *Locate* in your new applications. You may see performance gains in existing applications if you convert them to use the new methods.

Searching for records based on indexed fields

Table components support a set of *Goto* search methods for backward compatibility. *Goto* methods enable you to search for a record based on indexed fields, referred to as a *key*, and make the first record found the new current record.

For Paradox and dBASE tables, the key must always be an index, which you can specify in a table component's *IndexName* property. For SQL tables, the key can also be a list of fields you specify in the *IndexFieldNames* property. You can also specify a field list for Paradox or dBASE tables, but the fields must have indexes defined on them. For more information about *IndexName* and *IndexFieldNames*, see "Searching on alternate indexes" on page 20-8.

Tip To search on nonindexed fields in a Paradox or dBASE table, use *Locate*. Alternatively, you can use a *TQuery* component and a SELECT statement to search on nonindexed fields in Paradox or dBASE fields. For more information about *TQuery*, see Chapter 21, "Working with queries."

The following table summarizes the six related *Goto* and *Find* methods an application can use to search for a record:

Table 20.3 Legacy TTable search methods

Method	Purpose
<i>EditKey</i>	Preserves the current contents of the search key buffer and puts the table into <i>dsSetKey</i> state so your application can modify existing search criteria prior to executing a search.
<i>FindKey</i>	Combines the <i>SetKey</i> and <i>GotoKey</i> methods in a single method.
<i>FindNearest</i>	Combines the <i>SetKey</i> and <i>GotoNearest</i> methods in a single method.
<i>GotoKey</i>	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.
<i>GotoNearest</i>	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
<i>SetKey</i>	Clears the search key buffer and puts the table into <i>dsSetKey</i> 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.

Executing a search with *Goto* methods

To execute a search using *Goto* methods, follow these general steps:

- 1 Specify the index to use for the search in the *IndexName* property, if necessary. (For SQL tables, list the fields to use as a key in *IndexFieldNames* instead.) If you use a table's primary index, you do not need to set these properties.

- 2 Open the table.
- 3 Put the table in *dsSetKey* state with *SetKey*.
- 4 Specify the value(s) to search on in the *Fields* property. *Fields* is a string list that you index into with ordinal numbers corresponding to columns. The first column number in a table 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, moves to the first record containing a field value that exactly matches the text in an edit box on a form:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  Table1.SetKey;
  Table1.Fields[0].AsString := Edit1.Text;
  if not Table1.GotoKey then
    ShowMessage('Record not found');
end;
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

If a record exists with “Sm” as the first two characters, 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

To execute a search using *Find* methods, follow these general steps:

- 1 Specify the index to use for the search in the *IndexName* property, if necessary. (For SQL tables, list the fields to use as a key in *IndexFieldNames* instead.) If you use a table's primary index, you do not need to set these properties.
- 2 Open the table.
- 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 for a table component 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 a table 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 a table has a three column primary key, and you want to search for values in just the first column, set *KeyFieldCount* to 1.

For tables 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 alternate indexes

If you want to search on an index other than the primary index for a table, then you must specify the name of the index to use in the *IndexName* property for the table. For example, if the CUSTOMER table had a secondary index named “CityIndex” on which you wanted to search for a value, you need to set the value of the table’s *IndexName* property to “CityIndex”:

```
Table1.Close;
Table1.IndexName := 'CityIndex';
Table1.Open;
Table1.SetKey;
Table1['City'] := Edit1.Text;
Table1.GotoNearest;
```

Instead of specifying an index name, you can list fields to use as a key in the *IndexFieldNames* property. For Paradox and dBASE tables, the fields you list must be indexed, or an exception is raised when you execute the search. For SQL tables, the fields you list need not be indexed.

Repeating or extending a search

Each time you call *SetKey* or *FindKey* it 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, if the “CityIndex” index includes both the *City* and *Country* fields, to find a record with a specified company name in a specified city, use the following code:

```
Table1.EditKey;
Table1['Country'] := Edit2.Text;
Table1.GotoNearest;
```


Sorting records

An index determines the display order of records in a table. In general, records appear in ascending order based on a primary index (for dBASE tables without a primary index, sort order is based on physical record order). 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 (SQL only).

Specifying a different sort order requires the following steps:

- 1 Determining available indexes.
- 2 Specifying the alternate index or column list to use.

Retrieving a list of available indexes with `GetIndexNames`

At runtime, your application can call the `GetIndexNames` method to retrieve a list of available indexes for a table. `GetIndexNames` returns a string list containing valid index names. For example, the following code determines the list of indexes available for the `CustomersTable` dataset:

```
var
  IndexList: TList;
  :
  CustomersTable.GetIndexNames(IndexList);
```

Note For Paradox tables, the primary index is unnamed, and is therefore not returned by `GetIndexNames`. If you need to return to using a primary index on a Paradox table after using an alternative index, set the table's `IndexName` property to a null string.

Specifying an alternative index with `IndexName`

To specify that a table should be sorted using an alternative index, specify the index name in the table component's `IndexName` property. At design time you can specify this name in the Object Inspector, and at runtime you can access the property in your code. For example, the following code sets the index for `CustomersTable` to `CustDescending`:

```
CustomersTable.IndexName := 'CustDescending';
```

Specifying a dBASE index file

For dBASE tables that use non-production indexes, set the `IndexFiles` property to the name of the index file(s) to use before you set `IndexName`. At design time you can click the ellipsis button in the `IndexFiles` property value in the Object Inspector to invoke the Index Files editor.

To see a list of available index files, choose Add, and select one or more index files from the list. A dBASE index file can contain multiple indexes. To select an index from the index file, select the index name from the *IndexName* drop-down list in the Object Inspector. You can also specify multiple indexes in the file by entering desired index names, separated by semicolons.

You can also set *IndexFiles* and *IndexName* at runtime. For example, the following code sets the *IndexFiles* for the *AnimalsTable* table component to ANIMALS.MDX, and then sets *IndexName* to NAME:

```
AnimalsTable.IndexFiles := 'ANIMALS.MDX';
AnimalsTable.IndexName := 'NAME';
```

Specifying sort order for SQL tables

In SQL, sort order of rows is determined by the ORDER BY clause. You can specify the index used by this clause either with the

- *IndexName* property, to specify an existing index, or
- *IndexFieldNames* property, to create a pseudo-index based on a subset of columns in the table.

IndexName and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other. To use *IndexName*, see “Searching on alternate indexes” on page 20-8.

Specifying fields with IndexFieldNames

IndexFieldNames is a string list property. 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*:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Note If you use *IndexFieldNames* on Paradox and dBASE tables, Delphi attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

Examining the field list for an index

When your application uses an index at runtime, it can examine the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list of column names that comprise the index.

IndexFields is a string list containing the column names for the index. The following code fragment illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```

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];
      end;
    end;
  end;
end;

```

Note *IndexFieldCount* is not valid for a base table opened on an expression index.

Working with a subset of data

Production tables can be huge, so applications often need to limit the number of rows with which they work. For table components, there are two ways to limit records used by an application: ranges and filters. Filters can be used with any kind of dataset, including *TTable*, *TQuery*, and *TStoredProc* components. Because they apply to all datasets, you can find a full discussion of using filters in Chapter 18, “Understanding datasets.”

Ranges only apply to *TTable* components. Despite their similarities, ranges and filters have different uses. The following section discusses the differences between ranges and filters and then discusses how to use ranges.

Understanding the differences between ranges and filters

Both ranges and filters restrict visible records in a table 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, ranges can only be applied to indexed Paradox and dBASE tables (for SQL tables, ranges can be applied to any fields you list in the *IndexFieldNames* property). Ranges can only be ordered based on existing indexes.

A filter, on the other hand, is any set of contiguous and noncontiguous 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 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 very 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 a query component to select data for viewing and editing. For more information about using filters, see Chapter 18, “Understanding datasets.” For more information about using queries, see Chapter 21, “Working with queries.”

Creating and applying a new range

The process for creating and applying a range involves these general steps:

- 1 Putting the dataset into *dsSetKey* state and setting the starting index value for the range.
- 2 Setting the ending index value for the range.
- 3 Applying the range to the dataset.

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 be indexed fields when using Paradox and dBASE.

For example, suppose your application uses a table 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. If so, the following code could be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo') := StartVal.Text;
  SetRangeEnd;
  if EndVal.Text <> '' then
    FieldByName('CustNo') := EndVal.Text;
  ApplyRange;
end
```

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 table will be included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records will be 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 set more values than there are fields in the index, the extra fields are ignored when computing the range.

To finish specifying the start of a range, call *SetRangeEnd* or *ApplyRange*. These methods are discussed in the following sections.

Tip To start at the beginning of the dataset, omit the call to *SetRangeStart*.

You can also set the starting (and ending) values for a range with a call to the *SetRange* procedure. For more information about *SetRange*, see “Setting start- and end-range values” on page 20-13.

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 be indexed fields for Paradox and dBASE tables.

Note 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,

```
with Table1 do
begin
  SetRangeStart;
  FieldByName('LastName') := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName') := Edit2.Text;
  ApplyRange;
end;
```

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 set more values than there are fields in the index, an exception is raised.

To finish specifying the end of a range, call *ApplyRange*. For more information about applying a range, see “Applying a range” on page 20-14.

Note You can also set the ending (and starting) values for a range with a call to the *SetRange* procedure. *SetRange* is described in the next section.

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 statements establish a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

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. If you set more values than there are fields in the index, the extra fields are ignored when computing the range.

Note 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 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:

```
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Zzzzzz';
Table1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith.” The value specification could also be:

```
Table1['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to “Sm.” The following statement includes records with a *LastName* greater than or equal to “Smith” and a *FirstName* greater than or equal to “J”:

```
Table1['LastName'] := 'Smith';
Table1['FirstName'] := 'J';
```

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 table component to *True* to exclude records equal to ending range. For example,

```
KeyExclusive := True;
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Tyler';
Table1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

Applying a range

The *SetRange* methods described in the preceding sections establish the boundary conditions for a range, but do not put it into effect. To make a range take effect, call the *ApplyRange* procedure. *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:

```

:
Table1.CancelRange;
:
{later on, use the same range again. No need to call SetRangeStart, etc.}
Table1.ApplyRange;
:

```

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.

The process for editing and applying a range involves these general steps:

- 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. Fields specified must be indexed fields when using Paradox and dBASE.

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. For more information about ranges based on partial keys, see “Specifying a range based on partial keys” on page 20-14.

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. Fields must be indexed fields when using Paradox and dBASE.

Note 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.

Deleting all records in a table

To delete all rows of data in a table, call a table component's *EmptyTable* method at runtime. For SQL tables, this method only succeeds if you have DELETE privilege for the table. For example, the following statement deletes all records in a dataset:

```
PhoneTable.EmptyTable;
```

Caution Data you delete with *EmptyTable* is gone forever.

Deleting a table

At design time, to delete a table from a database, right-click the table component and select Delete Table from the context menu. The Delete Table menu pick will only be present if the table component represents an existing database table (the *DatabaseName* and *TableName* properties specify an existing table).

To delete a table at runtime, call the table component's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

Caution When you delete a table with *DeleteTable*, the table and all its data are gone forever.

Renaming a table

To rename a Paradox or dBASE table at design time, right-click the table component and select Rename Table from the context menu. You can also rename a table by typing over the name of an existing table next to the *TableName* property in the Object Inspector. When you change the *TableName* property, a dialog appears asking you if you want to rename the table. At this point, you can either choose to rename the table, or you can cancel the operation, changing the *TableName* property (for example, to create a new table) without changing the name of the table represented by the old value of *TableName*.

To rename a Paradox or dBASE table at runtime, call the table component's *RenameTable* method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable('CustInfo');
```


Creating a table

You can create new database tables at design time or at runtime. The Create Table command (at design time) or the *CreateTable* method (at runtime) provides a way to create tables without requiring SQL knowledge. They do, however, require you to be intimately familiar with the properties, events, and methods common to dataset components, *TTable* in particular. This is so that you can first define the table you want to create by doing the following:

- Set the *DatabaseName* property to the database that will contain the new table.
- 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*.
- Set the *TableName* property to the name of the new table. If the value of the *TableType* property is *ttParadox* or *ttDBase*, you do not need to specify an extension.
- Add field definitions to describe the fields in the new table. At design time, you can add the field definitions by double-clicking 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.
- Optionally, add index definitions that describe the desired indexes of the new table. At design time, you can add index definitions by double-clicking 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 the 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.

Note At design time, you can preload the field definitions and index definitions of an existing table into the *FieldDefs* and *IndexDefs* properties, respectively. 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, cancelling any prompts to rename the existing table. If you want to store these definitions with the table component (for example, if your application will be using them to create tables on user's systems), set the *StoreDefs* property to *True*.

Once the table is fully described, you are ready to create it. At design time, right-click the table component and choose Create Table. At runtime, call the *CreateTable* method to generate the specified table.

Warning 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.

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:

```

var
  NewTable: TTable;
  NewIndexOptions: TIndexOptions;
  TableFound: Boolean;
begin
  NewTable := TTable.Create;
  NewIndexOptions := [ixPrimary, ixUnique];
  with NewTable do
    begin
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      FieldDefs.Clear;
      FieldDefs.Add(Edit2.Text, ftInteger, 0, False);
      FieldDefs.Add(Edit3.Text, ftInteger, 0, False);
      IndexDefs.Clear;
      IndexDefs.Add('PrimaryIndex', Edit2.Text, NewIndexOptions);
    end;
  {Now check for prior existence of this table}
  TableFound := FindTable(Edit1.Text); {code for FindTable not shown}
  if TableFound = True then
    if MessageDlg('Overwrite existing table ' + Edit1.Text + '?', mtConfirmation,
      mbYesNo, 0) = mrYes then
      TableFound := False;
  if not TableFound then
    CreateTable; { create the table}
  end;
end;

```

Importing data from another table

You can use a table component's *BatchMove* method 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. Table 20.4 describes the possible settings for the mode specification:

Table 20.4 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 the same 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 their counterparts in the source table.

For example, the following code updates records in the current table with records from the *Customer* table:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove returns the number of records it imports successfully.

Caution Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

BatchMove performs only some of the functions available to your application directly through the *TBatchMove* component. If you need to move a large amount of data between or among tables, use the batch move component instead of calling a table's *BatchMove* function. For more information about using *TBatchMove*, see the following topic, "Using TBatchMove."

Using TBatchMove

TBatchMove encapsulates Borland Database Engine (BDE) features that enable 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.

Creating a batch move component

To create a batch move component:

- 1 Place the 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 component to which to move records (called the *Destination* dataset) on a form or in a data module.
- 3 Place a *TBatchMove* component from the Data Access page of the Component palette in a data module or on a 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 to copy, append, or update from. 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, or add a new table component for the destination.

If you are appending, updating, or deleting, *Destination* must be an existing table.

If you are copying a table and *Destination* is an existing table, then 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 more information about these modes, see “Specifying a batch move mode” on page 20-21.
- 7 Optionally set the *Transliterate* property. If *Transliterate* is *True* (the default), character data is transliterated 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 column based on their position in the source and destination tables. For more information about mapping columns, see “Mapping data types” on page 20-22.
- 9 Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch move 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” on page 20-23.

Specifying a batch move mode

The *Mode* property specifies the operation a batch move component performs:

Table 20.5 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

To append data, the destination dataset must already exist. 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

To update data, the destination dataset must already exist 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

To append and update data the destination dataset must already exist 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

To make a copy of a source dataset, the destination dataset should not already exist. If it does, the batch move operation overwrites the existing destination dataset 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 or through the SQL Explorer as appropriate.

Deleting

To delete data in the destination dataset, it must already exist 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” on page 20-23.

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:

```
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.

Synchronizing tables linked to the same database table

If more than one table component is linked to the same database table through their *DatabaseName* and *TableName* properties and the tables do not share a data source component, then each table has its own view on the data and its own current record. As users access records through each table component, the components' current records will differ.

You can force the current record for each of these table components to be the same with the *GotoCurrent* method. *GotoCurrent* sets its own table's current record to the current record of another table component. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent (CustomerTableTwo);
```

Tip If your application needs to synchronize table components in this manner, put the components in a data module and include the header for the data module in each unit that accesses the tables.

If you must synchronize table components on separate forms, you must include one form's header file in the source unit of the other form, and you must qualify at least one of the table names with its form name.

For example:

```
CustomerTableOne.GotoCurrent (Form2.CustomerTableTwo);
```

Creating master/detail forms

A table component's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two tables.

The *MasterSource* property is used to specify a data source from which the table will get data for the master table. For instance, if you link two tables in a master/detail

relationship, then the detail table can track the events occurring in the master table by specifying the master table's data source component in this property.

The *MasterFields* property specifies the column(s) common to both tables used to establish the link. To link tables based on multiple column names, use a semicolon delimited list:

```
Table1.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two tables, you can use the Field Link designer. For more information about the Field Link designer, see the *User's Guide*.

Building an example master/detail form

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*.

- 1 Place two *TTable* and two *TDataSource* components in a data module.
- 2 Set the properties of the first *TTable* component as follows:
 - *DatabaseName*: DBDEMOS
 - *TableName*: CUSTOMER
 - *Name*: CustomersTable
- 3 Set the properties of the second *TTable* component as follows:
 - *DatabaseName*: DBDEMOS
 - *TableName*: ORDERS
 - *Name*: OrdersTable
- 4 Set the properties of the first *TDataSource* component as follows:
 - *Name*: CustSource
 - *DataSet*: CustomersTable
- 5 Set the properties of the second *TDataSource* component as follows:
 - *Name*: OrdersSource
 - *DataSet*: OrdersTable
- 6 Place two *TDBGrid* components on a form.
- 7 Choose File | Include Unit Hdr to specify that the form should use the data module.
- 8 Set the *DataSource* property of the first grid component to "DataModule2->CustSource", and set the *DataSource* property of the second grid to "DataModule2->OrdersSource".
- 9 Set the *MasterSource* property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).
- 10 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.
- 11 Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.
- 12 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.

Working with nested tables

A nested table component provides access to data in a nested dataset of a table. The *NestedDataSet* property of a persistent nested dataset field contains a reference to the nested dataset. Since *TNestedDataSet* descends from *TBDEDataSet*, a nested table inherits BDE functionality, and so uses the Borland Database Engine (BDE) to access the nested table data. A nested table provides much of the functionality of a table component, but the data it accesses is stored in a nested table.

Setting up a nested table component

The following steps are general instructions for setting up a nested table component at design time. A table or live query component must be available that accesses a dataset containing a dataset or reference field. A persistent field for the *TDataSetField* or *TReferenceField* must also already exist. See “Working with dataset fields” on page 19-26.

To use a nested table component,

- 1 Place a nested table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
- 2 Set the *DataSetField* of the component to the name of the persistent dataset field or reference field to access. You can select fields from the drop-down list.
- 3 Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the nested table component. The data source component is used to pass a result set from the table to data-aware components for display.

Working with queries

This chapter describes the *TQuery* dataset component which enables you to use SQL statements to access data. It assumes you are familiar with the general discussion of datasets and data sources in Chapter 18, “Understanding datasets.”

A query component encapsulates an SQL statement that is used in a client application to retrieve, insert, update, and delete data from one or more database tables. SQL is an industry-standard relational database language that is used by most remote, server-based databases, such as Sybase, Oracle, InterBase, and Microsoft SQL Server. Query components can be used with remote database servers (if your version of Delphi includes SQL links), with Paradox, dBASE, FoxPro, and Access, and with ODBC-compliant databases.

Using queries effectively

To use the query component effectively, you must be familiar with

- SQL and your server’s SQL implementation, including limitations and extensions to the SQL-92 standard.
- The Borland Database Engine (BDE).

If you are an experienced desktop database developer moving to server-based applications, see “Queries for desktop developers” on page 21-2 for an introduction to queries before reading the remainder of this chapter. If you are new to SQL, you may want to purchase one of the many fine third party books that cover 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.

If you are an experienced database server developer, but are new to building Delphi clients, then you are already familiar with SQL and your server, but you may be unfamiliar with the BDE. See “Queries for server developers” on page 21-3 for an introduction to queries and the BDE before reading the remainder of this chapter.

Queries for desktop developers

As a desktop developer you are already familiar with the basic table, record, and field paradigm used by Delphi and the BDE. You feel very comfortable using a *TTable* component to gain access to every field in every data record in a dataset. You know that when you set a table's *TableName* property, you specify the database table to access.

Chances are you have also used a *TTable*'s range methods and filter property to limit the number of records available at any given time in your applications. Applying a range temporarily limits data access to a block of contiguously indexed records that fall within prescribed boundary conditions, such as returning all records for employees whose last names are greater than or equal to "Jones" and less than or equal to "Smith". Setting a filter temporarily restricts data access to a set of records that is usually noncontiguous and that meets filter criteria, such as returning only those customer records that have a California mailing address.

A query behaves in many ways very much like a table filter, except that you use the query component's *SQL* property (and sometimes the *Params* property) to identify the records in a dataset to retrieve, insert, delete, or update. In some ways a query is even more powerful than a filter because it lets you access

- More than one table at a time (called a "join" in SQL).
- A specified subset of rows *and* columns in its underlying table(s), rather than always returning all rows and columns. This improves both performance and security. Memory is not wasted on unnecessary data, and you can prevent access to fields a user should not view or modify.

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 by the BDE 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.

Most often you use queries to select the data that a user should see in your application, just as you do when you use a table component. Queries, however, can also perform update, insert, and delete operations instead of retrieving records for display. When you use a query to perform insert, update, and delete operations, the query ordinarily does not return records for viewing. In this way a query differs from a table.

To learn more about using the *SQL* property to write an SQL statement, see "Specifying the SQL statement to execute" on page 21-5. To learn more about using parameters in your SQL statements, see "Setting parameters" on page 21-8. To learn about executing a query, see "Executing a query" on page 21-12.

Queries for server developers

As a server developer you are already familiar with SQL and with the capabilities of your database server. To you a query is the SQL statement you use to access data. You know how to use and manipulate this statement and how to use optional parameters with it.

The SQL statement and its parameters are the most important parts of a query component. The query component's *SQL* property is used to provide the SQL statement to use for data access, and the component's *Params* property is an optional array of parameters to bind into the query. However, a query component is much more than an SQL statement and its parameters. A query component is also the interface between your client application and the BDE.

A client application uses the properties and methods of a query component to manipulate an SQL statement and its parameters, to specify the database to query, to prepare and unprepare queries with parameters, and to execute the query. A query component's methods call the BDE, which, in turn, processes your query requests, and communicates with the database server, usually through an SQL Links driver. The server passes a result set, if appropriate, back to the BDE, and the BDE returns it to your application through the query component.

When you work with a query component, you should be aware that some of the terminology used to describe BDE features can be confusing at first because it has very different meanings than what you are familiar with as an SQL database programmer. For example, the BDE commonly uses the term "alias" to refer to a shorthand name for the path to the database server. The BDE alias is stored in a configuration file, and is set in the query component's *DatabaseName* property. (You can still use table aliases, or "table correlation names," in your SQL statements.)

Similarly, the BDE help documentation, available online in \Borland\Common Files\BDE\BDE32.HLP, often refers to queries that use parameters as "parameterized queries" where you are more likely to think of SQL statements that use bound variables or parameter bindings.

Note This chapter uses BDE terminology because you will encounter it throughout Borland's documentation. Whenever confusion may result from using BDE terms, explanations are provided.

To learn more about using the *SQL* property to write an SQL statement, see "Specifying the SQL statement to execute" on page 21-5. To learn more about using parameters in your SQL statements, see "Setting parameters" on page 21-8. To learn about preparing a query, see "Preparing a query" on page 21-13, and to learn more about executing a query, see "Executing a query" on page 21-12.

What databases can you access with a query component?

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 (depending on your version of Delphi). You must have installed 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.

Delphi also supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you create a heterogeneous query, the BDE uses Local SQL to process the query. For more information, see "Creating heterogeneous queries" on page 21-14.

Using a query component: an overview

To use a query component in an application, follow these steps at design time:

- 1 Place a query component from the Data Access tab of the Component palette in a data module, and set its *Name* property appropriately for your application.
- 2 Set the *DatabaseName* property of the component to the name of the database to query. *DatabaseName* can be a BDE alias, an explicit directory path (for local tables), or the value from the *DatabaseName* property of a *TDatabase* component in the application.
- 3 Specify an SQL statement in the *SQL* property of the component, and optionally specify any parameters for the statement in the *Params* property. For more information, see "Specifying the SQL property at design time" on page 21-6.
- 4 If the query data is to be used with visual data controls, place a data source component from the Data Access tab of the Component palette in the data module, and set its *DataSet* property to the name of the query component. The data source component is used to return the results of the query (called a *result set*) from the query 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. For queries that only perform an action on a table and return no result set, use the *ExecSQL* method.

To execute a query for the first time at runtime, follow these steps:

- 1 Close the query component.
- 2 Provide an SQL statement in the *SQL* property if you did not set the *SQL* property at design time, or if you want to change the SQL statement already provided. To use the design-time statement as is, skip this step. For more information about setting the *SQL* property, see “Specifying the SQL statement to execute” on page 21-5.
- 3 Set parameters and parameter values in the *Params* property either directly or by using the *ParamByName* method. If a query does not contain parameters, or the parameters set at design time are unchanged, skip this step. For more information about setting parameters, see “Setting parameters” on page 21-8.
- 4 Call *Prepare* to initialize the BDE and bind parameter values into the query. Calling *Prepare* is optional, though highly recommended. For more information about preparing a query, see “Preparing a query” on page 21-13.
- 5 Call *Open* for queries that return a result set, or call *ExecSQL* for queries that do not return a result set. For more information about opening and executing a query see “Executing a query” on page 21-12.

After you execute a query for the first time, then as long as you do not modify the SQL statement, an application can repeatedly close and reopen or re-execute a query without preparing it again. For more information about reusing a query, see “Executing a query” on page 21-12.

Specifying the SQL statement to execute

Use the *SQL* property to specify the SQL query statement to execute. At design time a query is prepared and executed automatically when you set the query component’s *Active* property to *true*. At runtime, a query is prepared with a call to *Prepare*, and executed when the application calls the component’s *Open* or *ExecSQL* methods.

The *SQL* property is a *TStrings* object, which is an array of text strings and a set of properties, events, and methods that manipulate them. The strings in *SQL* are automatically concatenated to produce the SQL statement to execute. You can provide a statement in as few or as many separate strings as you desire. One advantage to using a series of strings is that you can divide the SQL statement into logical units (for example, putting the *WHERE* clause for a *SELECT* statement into its own string), so that it is easier to modify and debug a query.

The SQL statement can be a query that contains hard-coded field names and values, or it can be a parameterized query that contains replaceable parameters that represent field values that must be bound into the statement before it is executed. For example, this statement is hard-coded:

```
SELECT * FROM Customer WHERE CustNo = 1231
```

Hard-coded statements are useful when applications execute exact, known queries each time they run. At design time or runtime you can easily replace one hard-code

query with another hard-coded or parameterized query as needed. Whenever the SQL property is changed the query is automatically closed and unprepared.

Note In queries made against the BDE engine using local SQL, when column names in a query contain spaces or special characters, the column name must be enclosed in quotes and must be preceded by a table reference and a period. For example, BIOLIFE."Species Name". For information about valid column names, see your server's documentation.

A parameterized query contains one or more placeholder parameters, application variables that stand in for comparison values such as those found in the WHERE clause of a SELECT statement. Using parameterized queries enables you to change the value without rewriting the application. 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.

This statement is a parameterized query:

```
SELECT * FROM Customer WHERE CustNo = :Number
```

The variable *Number*, indicated by the leading colon, is a parameter that fills in for a comparison value that must be provided at runtime and that may vary each time the statement is executed. The actual value for *Number* is provided in the query component's *Params* property.

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 ensures that if a query uses its *DataSource* property to provide values for parameters, it can match the variable name to valid field names.

Specifying the SQL property at design time

You can specify the SQL property at design time using the String List editor. To invoke the String List editor for the SQL property:

- Double-click on the SQL property value column, or
- Click its ellipsis button.

You can enter an SQL statement in as many or as few lines as you want. Entering a statement on multiple lines, however, makes it easier to read, change, and debug. Choose OK to assign the text you enter to the SQL property.

Normally, the SQL property can contain only one complete SQL statement at a time, although these statements 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 in the SQL property.

Note With some versions of Delphi, 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 a query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use the SQL Builder, open it and use its online help.

Specifying an SQL statement at runtime

There are three ways to set the *SQL* property at runtime. An application can set the *SQL* property directly, it can call the *SQL* property's *LoadFromFile* method to read an SQL statement from a file, or an SQL statement in a string list object can be assigned to the *SQL* property.

Setting the SQL property directly

To directly set the *SQL* property at runtime,

- 1 Call *Close* to deactivate the query. Even though an attempt to modify the *SQL* property automatically deactivates the query, it is a good safety measure to do so explicitly.
- 2 If you are replacing the whole SQL statement, call the *Clear* method for the *SQL* property to delete its current SQL statement.
- 3 If you are building the whole SQL statement from nothing or adding a line to an existing statement, call the *Add* method for the *SQL* property to insert and append one or more strings to the *SQL* property to create a new SQL statement. If you are modifying an existing line use the *SQL* property with an index to indicate the line affected, and assign the new value.
- 4 Call *Open* or *ExecSQL* to execute the query.

The following code illustrates building an entire SQL statement from nothing.

```
with CustomerQuery do begin
    Close;                               { close the query if it's active }
    with SQL do begin
        Clear;                            { delete the current SQL statement, if any }
        Add('SELECT * FROM Customer');    { add first line of SQL... }
        Add('WHERE Company = "Sight Diver"'); { ... and second line }
    end;
    Open;                                 { activate the query }
end;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the *WHERE* clause already exists on the second line of the statement. It is referenced via the *SQL* property using an index of 1.

```
CustomerQuery.SQL[1] := 'WHERE Company = "Kauai Dive Shoppe";
```

Note If a query uses parameters, you should also set their initial values and call the *Prepare* method before opening or executing a query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from a file

You can also use the *LoadFromFile* method to assign an SQL statement in a text file to the SQL property. The *LoadFromFile* method automatically clears the current contents of the SQL property before loading the new statement from file. For example:

```
CustomerQuery.Close;
CustomerQuery.SQL.LoadFromFile('c:\orders.txt');
CustomerQuery.Open;
```

Note If the SQL statement contained in the file is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from string list object

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. For example, copying an SQL statement from a *TMemo* component:

```
CustomerQuery.Close;
CustomerQuery.SQL.Assign(Memo1.Lines);
CustomerQuery.Open;
```

Note If the SQL statement is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Setting parameters

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. Before a parameterized query is executed for the first time, your application should call the *Prepare* method to bind the current values for the parameters to the SQL statement. Binding means that the BDE and the server allocate resources for the statement and its parameters that improve the execution speed of the query.

```
with Query1 do begin
  Close;
  Unprepare;
  ParamByName('Name').AsString := 'Belize';
```

```

ParamByName('Capital').AsString := 'Belmopan';
ParamByName('Population').AsInteger := '240000';
Prepare;
Open;
end;

```

Supplying parameters at design time

At design time, parameters in the SQL statement appear in the parameter collection editor. To access the *TParam* objects for the parameters, invoke the parameter collection editor, select a parameter, and access the *TParam* properties in the Object Inspector. If the SQL statement does not contain any parameters, no *TParam* objects are listed in the collection editor. You can only add parameters by writing them in the SQL statement.

To access parameters,

- 1 Select the query component.
- 2 Click on the ellipsis button for the *Params* property in Object Inspector.
- 3 In the parameter collection editor, select a parameter.
- 4 The *TParam* object for the selected parameter appears in the Object Inspector.
- 5 Inspect and modify the properties for the *TParam* in the Object Inspector.

For queries that do not already contain parameters (the SQL property is empty or the existing SQL statement has no parameters), the list of parameters in the collection editor dialog is empty. If parameters are already defined for a query, then the parameter editor lists all existing parameters.

Note The *TQuery* component shares the *TParam* object and its collection editor with a number of different components. While the right-click context menu of the collection editor always contains the Add and Delete options, they are never enabled for *TQuery* parameters. The only way to add or delete *TQuery* parameters is in the SQL statement itself.

As each parameter in the collection editor is selected, the Object Inspector displays the properties and events for that parameter. Set the values for parameter properties and methods in the Object Inspector.

The *DataType* property lists the BDE data type for the parameter selected in the editing dialog. Initially the type will be *ftUnknown*. You must set a data type for each parameter. In general, BDE data types conform to server data types. For specific BDE-to-server data type mappings, see the BDE help in \Borland\Common Files\BDE\BDE32.HLP.

The *ParamType* property lists the type of parameter selected in the editing dialog. Initially the type will be *ptUnknown*. You must set a type for each parameter.

Use the *Value* property to specify a value for the selected parameter at design-time. This is not mandatory when parameter values are supplied at runtime. In these cases, leave *Value* blank.

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.
- *Params* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set. This method uses variants and avoids the need to cast values.

For all of the examples below, assume the *SQL* property contains the SQL statement below. All three parameters used are of data type *ftString*.

```
INSERT INTO "COUNTRY.DB"
(Name, Capital, Continent)
VALUES (:Name, :Capital, :Continent)
```

The following code uses *ParamByName* to assign the text of an edit box to the Capital parameter:

```
Query1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 1 (the Capital parameter is the second parameter in the SQL statement):

```
Query1.Params[1].AsString := Edit1.Text;
```

The command line below sets all three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Country;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Using a data source to bind parameters

If parameter values for a parameterized query are not bound at design time or specified at runtime, the query component attempts to supply values for them based on its *DataSource* property. *DataSource* specifies a different table or query component that the query component can search for field names that match the names of unbound parameters. This search dataset must be created and populated before you create the query component that uses it. If matches are found in the search dataset, the query component binds the parameter values to the values of the fields in the current record pointed to by the data source.

You can create a simple application to understand how to use the *DataSource* property to link a query in a master-detail form. Suppose the data module for this application is called, *LinkModule*, and that it contains a query component called *OrdersQuery* that has the following *SQL* property:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = :CustNo
```

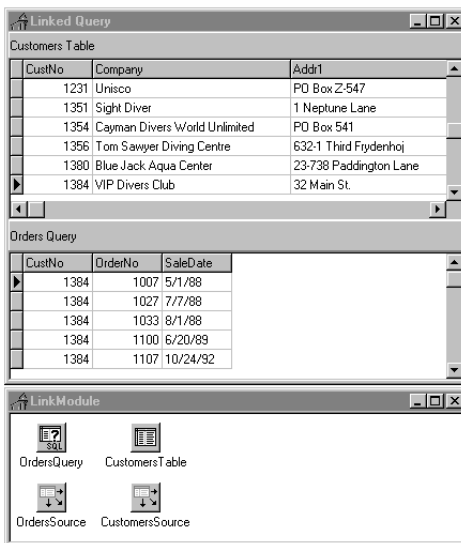
The *LinkModule* data module also contains:

- A *TTable* dataset component named *CustomersTable* linked to the CUSTOMER.DB table.
- A *TDataSource* component named *OrdersSource*. The *DataSet* property of *OrdersSource* points to *OrdersQuery*.
- A *TDataSource* named *CustomersSource* linked to *CustomersTable*. The *DataSource* property of the *OrdersQuery* component is also set to *CustomersSource*. This is the setting that makes *OrdersQuery* a linked query.

Suppose, too, that this application has a form, named *LinkedQuery* that contains two data grids, a *Customers Table* grid linked to *CustomersSource*, and an *Order Query* grid linked to *OrdersSource*.

The following figure illustrates how this application appears at design time.

Figure 21.1 Sample master/detail query form and data module at design time



Note If you build this application, create the table component and its data source before creating the query component.

If you compile this application, at runtime the `:CustNo` parameter in the SQL statement for *OrdersQuery* is not assigned a value, so *OrdersQuery* tries to match the parameter by name against a column in the table pointed to by *CustomersSource*. *CustomersSource* gets its data from *CustomersTable*, which, in turn, derives its data from the CUSTOMER.DB table. Because CUSTOMER.DB contains a column called “CustNo,” the value from the *CustNo* field in the current record of the *CustomersTable* dataset is assigned to the `:CustNo` parameter for the *OrdersQuery* SQL statement. The grids are linked in a master-detail relationship. At runtime, each time you select a different record in the Customers Table grid, the *OrdersQuery* SELECT statement executes to retrieve all orders based on the current customer number.

Executing a query

After you specify an SQL statement in the SQL property and set any parameters for the query, you can execute the query. When a query is executed, the BDE receives and processes SQL statements from your application. If the query is against local tables (dBASE and Paradox), the BDE SQL engine processes the SQL statement and, for a SELECT query, returns data to the application. If the query is against an SQL server table and the *TQuery.RequestLive* property is set to *False*, the SQL statement is not processed or interpreted by the BDE, but is passed directly to the database system. If the *RequestLive* property is set to *True*, the SQL statement will need to abide by local SQL standards as the BDE will need to use it for conveying data changes to the table.

Note Before you execute a query for the first time, you may want to call the *Prepare* method to improve query performance. *Prepare* initializes the BDE and the database server, each of which allocates system resources for the query. For more information about preparing a query, see “Preparing a query” on page 21-13.

The following sections describe executing both static and dynamic SQL statements at design time and at runtime.

Executing a query at design time

To execute a query at design time, set its *Active* property to *true* in the Object Inspector.

The results of the query, if any, are displayed in any data-aware controls associated with the query component.

Note The *Active* property can be used only with queries that returns a result set, such as by the SELECT statement.

Executing a query at runtime

To execute a query at runtime, use one of the following methods:

- *Open* executes a query that returns a result set, such as with the SELECT statement.
- *ExecSQL* executes a query that does not return a result set, such as with the INSERT, UPDATE, or DELETE statements.

Note If you do not know at design time whether a query will return a result set at runtime, code both types of query execution statements in a **try..except** block. Put a call to the *Open* method in the **try** clause. This allows you to suppress the error message that would occur due to using an activate method not applicable to the type of SQL statement used. Check the type of exception that occurs. If it is other than an *ENoResult* exception, the exception occurred for another reason and must be processed. This works because an action query will be executed when the query is activated with the *Open* method, but an exception occurs in addition to that.

```

try
  Query2.Open;
except
  on E: Exception do
    if not (E is ENoResultSet) then
      raise;
end;

```

Executing a query that returns a result set

To execute a query that returns a result set (a query that uses a SELECT statement), follow these steps:

- 1 Call *Close* to ensure that the query is not already open. If a query is already open you cannot open it again without first closing it. Closing a query and reopening it fetches a new version of data from the server.
- 2 Call *Open* to execute the query.

For example:

```

CustomerQuery.Close;
CustomerQuery.Open; { query returns a result set }

```

For information on navigating within a result set, see “Disabling bi-directional cursors” on page 21-15. For information on editing and updating a result set, see “Working with result sets” on page 21-16.

Executing a query without a result set

To execute a query that does not return a result set (a query that has an SQL statement such as INSERT, UPDATE, or DELETE), call *ExecSQL* to execute the query.

For example:

```

CustomerQuery.ExecSQL; { query does not return a result set }

```

Preparing a query

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 BDE for parsing, resource allocation, and optimization. The BDE, in turn, notifies the database server to prepare for the query. The server, too, may allocate resources for the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by calling the *Prepare* method. If you do not prepare a query before executing it, then Delphi automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though Delphi prepares queries for you, it is better programming practice to prepare a query explicitly. That way your code is self-documenting, and your intentions are clear. For example:

```

CustomerQuery.Close;
if not (CustomerQuery.Prepared) then
  CustomerQuery.Prepare;
CustomerQuery.Open;

```

This example checks the query component's *Prepared* property to determine if a query is already prepared. *Prepared* is a Boolean value that is *true* if a query is already prepared. If the query is not already prepared, the example calls the *Prepare* method before calling *Open*.

Unpreparing a query to release resources

The *UnPrepare* method sets the *Prepared* property to *false*. *UnPrepare*

- Ensures that the *SQL* property is prepared prior to executing it again.
- Notifies the BDE to release the internal resources allocated for the statement.
- Notifies the server to release any resources it has allocated for the statement.

To unprepare a query, call

```
CustomerQuery.UnPrepare;
```

Note When you change the text of the *SQL* property for a query, the query component automatically closes and unprepares the query.

Creating heterogeneous queries

Delphi supports *heterogeneous queries*, that is, queries made against tables in more than one database. A heterogeneous query may join tables on different servers, and even different types of servers. For example, a heterogeneous query might involve a table in a Oracle database, a table in a Sybase database, and a local dBASE 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, follow these steps:

- 1 Define separate BDE aliases for each database accessed in the query. Leave the *DatabaseName* property of the *TQuery* blank; the names of the two databases used will be specified in the SQL statement.
- 2 Specify the SQL statement to execute in the *SQL* property. Precede each table name in the SQL statement with the BDE alias for the database where that table can be found. The table reference is preceded by the name of the BDE alias, enclosed in colons. This whole reference is then enclosed in quotation marks.
- 3 Set any parameters for the query in the *Params* property.
- 4 Call *Prepare* to prepare the query for execution prior to executing it for the first time.
- 5 Call *Open* or *ExecSQL* depending on the type of query to execute.

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.

Improving query performance

Following are steps you can take to improve query execution speed:

- Set a query's *UniDirectional* property to *true* if you do not need to navigate backward through a result set (SQL-92 does not, itself, permit backward navigation through a result set). By default, *UniDirectional* is *false* because the BDE supports bi-directional cursors by default.
- Prepare the query before execution. This is especially helpful when you plan to execute a single query several times. You need only prepare the query once, before its first use. For more information about query preparation, see "Preparing a query" on page 21-13.

Disabling bi-directional cursors

The *UniDirectional* property determines whether or not BDE bi-directional cursors are enabled for a query. When a query 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.

UniDirectional is *false* by default, meaning that the cursor for a result set can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries. To improve query performance, you may be able to set *UniDirectional* to *true*, restricting a cursor to forward movement through a result set.

If you do not need to be able to navigate backward through a result set, you can set *UniDirectional* to *true* for a query. Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```

if not (CustomerQuery.Prepared) then begin
    CustomerQuery.UniDirectional := True;
    CustomerQuery.Prepare;
end;
CustomerQuery.Open; { returns a result set with a one-way cursor }

```

Working with result sets

By default, the result set returned by a query is read-only. Your application can display field values from the result set in data-aware controls, but users cannot edit those values. To enable editing of a result set, your application must request a “live” result set.

Enabling editing of a 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 or not a query uses the local SQL parser or a server’s SQL parser. Heterogeneous joins and queries executed against Paradox or dBASE are parsed by the BDE using local SQL. Queries against a remote database server are parsed by the server.

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *true*.

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 pass-through SQL queries made against a remote server.

Local SQL requirements for a live result set

For queries that use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. The local SQL parser is used when a query is made against one or more dBASE or Paradox tables, or one or more remote server tables when those table names in the query are preceded by a BDE database alias. The following sections describe the restrictions that must be met to return a live result set for local SQL.

Restrictions on live queries

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

Remote server SQL requirements for a live result set

For queries that use passthrough SQL, which includes all queries made solely against remote database servers, live result sets are restricted to the standards defined by SQL-92 and any additional, server-imposed restrictions.

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

Restrictions on updating a live result set

If a query returns a live result set, you may not be able to update the result set directly if the result set contains linked fields or you switch indexes before attempting an update. If these conditions exist, you may be able to treat the result set as a read-only result set, and update it accordingly.

Updating a read-only result set

Applications can update data returned in a read-only result set if they are using cached updates. To update a read-only result set associated with a query component:

- 1 Add a `TUpdateSQL` component to the data module in your application to essentially give you the ability to post updates to a read-only dataset.
- 2 Enter the SQL update statement for the result set to the update component's `ModifySQL`, `InsertSQL`, or `DeleteSQL` properties.
- 3 Set the query component's `CachedUpdate` property to `True`.

For more information about using cached updates, see Chapter 25, "Working with cached updates."

Working with stored procedures

This chapter describes how to use stored procedures in your database applications. A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. There are two fundamental types of stored procedures. The first type retrieves data (like with a `SELECT` query). The retrieved data can be in the form of a dataset consisting of one or more rows of data, divided into one or more columns. Or the retrieved data can be in the form of individual pieces of information. The second type does not return data, but performs an action on data stored in the database (like with a `DELETE` statement). Some database servers support performing both types of operations in the same procedure.

Stored procedures that return data do so in different ways, depending on how the stored procedure is composed and the database system used. Some, like InterBase, return all data (datasets and individual pieces of information) exclusively with output parameters. Others are capable of returning a cursor to data. And still others, like Microsoft SQL Server and Sybase, can return a dataset and individual pieces of information.

In Delphi applications, access to stored procedures is provided by the *TStoredProc* and *TQuery* components. The choice of which to use for the access is predicated on how the stored procedure is coded, how data is returned (if any), and the database system used. The *TStoredProc* and *TQuery* components are both descendants of *TDataSet*, and inherit behaviors from *TDataSet*. For more information about *TDataSet*, see Chapter 18, “Understanding datasets.”

A stored procedure component is used to execute stored procedures that do not return any data, to retrieve individual pieces of information in the form of output parameters, and to relay a returned dataset to an associated data source component (this last being database-specific). The stored procedure component allows values to be passed to and return from the stored procedure through parameters, each parameter defined in the *Params* property. The stored procedure component also provides a *GetResults* method to force the stored procedure to return a dataset (some database servers require this before a result set is produced). The stored procedure

component is the preferred means for using stored procedures that either do not return any data or only return data through output parameters.

A query component is primarily used to run stored procedures that return datasets. This includes InterBase stored procedures that only return datasets via output parameters. The query component can also be used to execute a stored procedure that does not return a dataset or output parameter values.

Use parameters to pass distinct values to or return values from a stored procedure. Input parameter values are used in such places as the WHERE clause of a SELECT statement in a stored procedure. An output parameter allows a stored procedure to pass a single value to the calling application. Some stored procedures return a result parameter. See the documentation for the specific database server you are using for details on the types of parameters that are supported and how they are used in the server's procedure language.

When should you use stored procedures?

If your server defines stored procedures, you should use them if they apply to the needs of your application. A database server developer creates stored procedures to handle frequently-repeated database-related tasks. Often, operations that act upon large numbers of rows in database tables—or that use aggregate or mathematical functions—are candidates for stored procedures. If stored procedures exist on the remote database server your application uses, you should take advantage of them in your application. Chances are you need some of the functionality they provide, and you stand to improve the performance of your database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result—a single value representing the standard deviation—it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

See your server's database documentation for more information about its support for stored procedures.

Using a stored procedure

How a stored procedure is used in a Delphi application 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.

In general terms, to access a stored procedure on a server, an application must:

- 1 Instantiate a *TStoredProc* component and optionally associate it with a stored procedure on the server. Or instantiate a *TQuery* component and compose the contents of its *SQL* property to perform either a SELECT query against the stored procedure or an EXECUTE command, depending on whether the stored procedure returns a result set. For more information about creating a *TStoredProc*, see “Creating a stored procedure component” on page 22-3. For more information about creating a *TQuery* component, see Chapter 21, “Working with queries”.
- 2 Provide input parameter values to the stored procedure component, if necessary. When a stored procedure component is not associated with stored procedure on a server, you must provide additional input parameter information, such as parameter names and data types. For more information about providing input parameter information, see “Setting parameter information at design time” on page 22-13.
- 3 Execute the stored procedure.
- 4 Process any result and output parameters. As with any other dataset component, you can also examine the result dataset returned from the server. For more information about output and result parameters, see “Using output parameters” on page 22-11 and “Using the result parameter” on page 22-12. For information about viewing records in a dataset, see “Using stored procedures that return result sets” on page 22-5.

Creating a stored procedure component

To create a stored procedure component for a stored procedure on a database server:

- 1 Place a stored procedure component from the Data Access page of the Component palette in a data module.
- 2 Optionally set the *DatabaseName* property of the stored procedure component to the name of the database in which the stored procedure is defined. *DatabaseName* must be a BDE alias or the same value as in the *DatabaseName* property of a *TDatabase* that can connect to the server.

Normally you should specify the *DatabaseName* property, but if the server database against which your application runs is currently unavailable, you can still create and set up a stored procedure component by omitting the *DatabaseName* and supplying a stored procedure name and input, output, and result parameters at design time. For more information about input parameters, see “Using input parameters” on page 22-10. For more information about output parameters, see “Using output parameters” on page 22-11. For more information about the result parameter, see “Using the result parameter” on page 22-12.

- 3 Optionally set the *StoredProcName* property to the name of the stored procedure to use. If you provided a value for the *DatabaseName* property, then you can select a stored procedure name from the drop-down list for the property. A single *TStoredProc* component can be used to execute any number of stored procedures by setting the *StoredProcName* property to a valid name in the application. It may not always be desirable to set the *StoredProcName* at design time.

- 4 Double-click the *Params* property value box to invoke the StoredProc Parameters editor to examine input and output parameters for the stored procedure. If you did not specify a name for the stored procedure in Step 3, or you specified a name for the stored procedure that does not exist on the server specified in the *DatabaseName* property in Step 2, then when you invoke the parameters editor, it is empty.

Not all servers return parameters or parameter information. See your server's documentation to determine what information about its stored procedures it returns to client applications.

Note If you do not specify the *DatabaseName* property in Step 2, then you must use the StoredProc Parameters editor to set up parameters at design time. For information about setting parameters at design time, see "Setting parameter information at design time" on page 22-13.

Creating a stored procedure

Ordinarily, stored procedures are created when the application and its database is created, using tools supplied by the database system vendor. However, it is possible to create stored procedures at runtime. The specific SQL statement used will vary from one database system to another because the procedure language varies so greatly. Consult the documentation for the specific database system you are using for the procedure language that is supported.

A stored procedure can be created by an application at runtime using an SQL statement issued from a *TQuery* component, typically with a CREATE PROCEDURE statement. If parameters are used in the stored procedure, set the *ParamCheck* property of the *TQuery* to *False*. This prevents the *TQuery* from mistaking the parameter in the new stored procedure from a parameter for the *TQuery* itself.

Note You can also use the SQL Explorer to examine, edit, and create stored procedures on the server.

After the SQL property has been populated with the statement to create the stored procedure, execute it by invoking the *ExecSQL* method.

```
with Query1 do begin
  ParamCheck := False;
  with SQL do begin
    Clear;
    Add('CREATE PROCEDURE GET_MAX_EMP_NAME');
    Add('RETURNS (Max_Name CHAR(15))');
    Add('AS');
    Add('BEGIN');
    Add(' SELECT MAX(LAST_NAME)');
    Add(' FROM EMPLOYEE');
    Add(' INTO :Max_Name;');
    Add(' SUSPEND;');
    Add('END');
  end;
  ExecSQL;
end;
```


Preparing and executing a stored procedure

To use a stored procedure, you can optionally prepare it, and then execute it.

You can prepare a stored procedure at:

- Design time, by choosing OK in the Parameters editor.
- Runtime, by calling the *Prepare* method of the stored procedure component.

For example, the following code prepares a stored procedure for execution:

```
StoredProc1.Prepare;
```

Note If your application changes parameter information at runtime, such as when using Oracle overloaded procedures, you should prepare the procedure again.

To execute a prepared stored procedure, call the *ExecProc* method for the stored procedure component. The following code illustrates code that prepares and executes a stored procedure:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.Prepare;
StoredProc1.ExecProc;
```

Note If you attempt to execute a stored procedure before preparing it, the stored procedure component 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 call *Prepare* yourself, and then only call *UnPrepare* once, when you no longer need to execute the procedure.

When you execute a stored procedure, it can return all or some of these items:

- A dataset consisting of one or more records that can be viewed in data-aware controls associated with the stored procedure through a data source component.
- Output parameters.
- A result parameter that contains status information about the stored procedure's execution.

To determine the return items to expect from a stored procedure on your server, see your server's documentation.

Using stored procedures that return result sets

Stored procedures that return data in datasets, rows and columns of data, should most often be used with a query component. However, with database servers that support returning a dataset by a stored procedure, a stored procedure component can serve this purpose.

Retrieving a result set with a TQuery

To retrieve a dataset from a stored procedure using a *TQuery* component:

- 1 Instantiate a query component.

- 2 In the *TQuery.SQL* property, write a SELECT query that uses the name of the stored procedure instead of a table name.
- 3 If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
- 4 Set the *Active* property to *True* or invoke the *Open* method.

For example, the InterBase stored procedure GET_EMP_PROJ, below, accepts a value using the input parameter EMP_NO and returns a dataset through the output parameter PROJ_ID.

```
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 SQL statement issued from a *TQuery* to use this stored procedure would be:

```
SELECT *
FROM GET_EMP_PROJ(52)
```

Retrieving a result set with a TStoredProc

To retrieve a dataset from a stored procedure using a *TStoredProc* component:

- 1 Instantiate a stored procedure component.
- 2 In the *StoredProcName* property, specify the name of the stored procedure.
- 3 If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
- 4 Set the *Active* property to *True* or invoke the *Open* method.

For example, the Sybase stored procedure GET_EMPLOYEES, below, accepts an input parameter named @EMP_NO and returns a result set based on that value.

```
CREATE PROCEDURE GET_EMPLOYEES @EMP_NO SMALLINT
AS SELECT EMP_NAME, EMPLOYEE_NO FROM EMPLOYEE_TABLE
WHERE (EMPLOYEE_NO = @EMP_NO)
```

The Delphi code to fill the parameter with a value and activate the stored procedure component is:

```
with StoredProc1 do begin
    Close;
    ParamByName('EMP_NO').AsSmallInt := 52;
    Active := True;
end;
```

Using stored procedures that return data using parameters

Stored procedures can be composed to retrieve individual pieces of information, as opposed to whole rows of data, through parameters. For instance, a stored procedure might retrieve the maximum value for a column, add one to that value, and then return that value to the application. Such stored procedures can be used and the values inspected using either a *TQuery* or a *TStoredProc* component. The preferred method for retrieving parameter values is with a *TStoredProc*.

Retrieving individual values with a TQuery

Parameter values retrieved via a *TQuery* component take the form of a single-row dataset, even if only one parameter is returned by the stored procedure. To retrieve individual values from stored procedure parameters using a *TQuery* component:

- 1 Instantiate a query component.
- 2 In the *TQuery.SQL* property, write a `SELECT` query that uses the name of the stored procedure instead of a table name. The `SELECT` clause of this query can specify the parameter by its name, as if it were a column in a table, or it can simply use the `*` operator to retrieve all parameter values.
- 3 If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
- 4 Set the *Active* property to *True* or invoke the *Open* method.

For example, the InterBase stored procedure `GET_HIGH_EMP_NAME`, below, retrieves the alphabetically last value in the `LAST_NAME` column of a table named `EMPLOYEE`. The stored procedure returns this value in the output parameter `High_Last_Name`.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
    SELECT MAX(LAST_NAME)
    FROM EMPLOYEE
    INTO :High_Last_Name;
    SUSPEND;
END
```

The SQL statement issued from a *TQuery* to use this stored procedure would be:

```
SELECT High_Last_Name
FROM GET_HIGH_EMP_NAME
```

Retrieving individual values with a TStoredProc

To retrieve individual values from stored procedure output parameters using a *TStoredProc* component:

- 1 Instantiate a stored procedure component.
- 2 In the *StoredProcName* property, specify the name of the stored procedure.

- 3 If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
- 4 Invoke the *ExecProc* method.
- 5 Inspect the values of individual output parameters using the *Params* property or *ParamByName* method.

For example, the InterBase stored procedure GET_HIGH_EMP_NAME, below, retrieves the alphabetically last value in the LAST_NAME column of a table named EMPLOYEE. The stored procedure returns this value in the output parameter High_Last_Name.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
SELECT MAX(LAST_NAME)
FROM EMPLOYEE
INTO :High_Last_Name;
SUSPEND;
END
```

The Delphi code to get the value in the High_Last_Name output parameter and store it to the *Text* property of a *TEdit* component is:

```
with StoredProc1 do begin
StoredProcName := 'GET_HIGH_EMP_NAME';
ExecProc;
Edit1.Text := ParamByName('High_Last_Name').AsString;
end;
```

Using stored procedures that perform actions on data

Stored procedures can be coded such that they do not return any data at all, and only perform some action in the database. SQL operations involving the INSERT and DELETE statements are good examples of this type of stored procedure. For instance, instead of allowing a user to delete a row directly, a stored procedure might be used to do so. This would allow the stored procedure to control what is deleted and also to handle any referential integrity aspects, such as a cascading delete of rows in dependent tables.

Executing an action stored procedure with a TQuery

To execute an action stored procedure using a *TQuery* component:

- 1 Instantiate a query component.
- 2 In the *TQuery.SQL* property, include the command necessary to execute the stored procedure and the stored procedure name. (The command to execute a stored procedure can vary from one database system to another. In InterBase, the command is EXECUTE PROCEDURE.)

- 3 If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
- 4 Invoke the *TQuery.ExecSQL* method.

For example, the InterBase stored procedure `ADD_EMP_PROJ`, below, adds a new row to the table `EMPLOYEE_PROJECT`. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
    BEGIN
        INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
        VALUES (:EMP_NO, :PROJ_ID);
        WHEN SQLCODE -530 DO
            EXCEPTION UNKNOWN_EMP_ID;
    END
    SUSPEND;
END
```

The SQL statement issued from a *TQuery* to execute this stored procedure would be:

```
EXECUTE PROCEDURE ADD_EMP_PROJ(20, "GUIDE")
```

Executing an action stored procedure with a TStoredProc

To retrieve individual values from stored procedure output parameters using a *TStoredProc* component:

- 1 Instantiate a stored procedure component.
- 2 In the *StoredProcName* property, specify the name of the stored procedure.
- 3 If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
- 4 Invoke the *ExecProc* method.

For example, the InterBase stored procedure `ADD_EMP_PROJ`, below, adds a new row to the table `EMPLOYEE_PROJECT`. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
    BEGIN
        INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
        VALUES (:EMP_NO, :PROJ_ID);
        WHEN SQLCODE -530 DO
            EXCEPTION UNKNOWN_EMP_ID;
    END
    SUSPEND;
END
```

The Delphi code to execute the `ADD_EMP_PROJ` stored procedure is:

```
with StoredProc1 do begin
    StoredProcName := 'ADD_EMP_PROJ';
    ExecProc;
end;
```

Understanding stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, 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 example, individual stored procedures on any server may either be implemented using input parameters, or may not be. 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 *TParam* objects in the *TStoredProc.Params* property. If the name of the stored procedure is specified at design time in the *StoredProcName* property, a *TParam* object is automatically created for each parameter and added to the *Params* property. If the stored procedure name is not specified until runtime, the *TParam* objects need to be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* objects allows a single *TStoredProc* component 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. For more information about displaying records in data-aware controls, see “Using stored procedures that return result sets” on page 22-5.

Using input parameters

Application use input parameters to pass singleton data values to a stored procedure. Such values are then used in SQL statements within the stored procedure, such as a

comparison value for a WHERE clause. If a stored procedure requires an input parameter, assign a value to the parameter prior to executing the stored procedure.

If a stored procedure returns a dataset and is used through a SELECT query in a *TQuery* component, supply input parameter values as a comma-separated list, enclosed in parentheses, following the stored procedure name. For example, the SQL statement below retrieves data from a stored procedure named GET_EMP_PROJ and supplies an input parameter value of 52.

```
SELECT PROJ_ID
FROM GET_EMP_PROJ(52)
```

If a stored procedure is executed with a *TStoredProc* component, use the *Params* property or the *ParamByName* method access to set each input parameter. Use the *TParam* property appropriate for the data type of the parameter, such as the *TParam.AsString* property for a CHAR type parameter. Set input parameter values prior to executing or activating the *TStoredProc* component. In the example below, the EMP_NO parameter (type SMALLINT) for the stored procedure GET_EMP_PROJ is assigned the value 52.

```
with StoredProc1 do begin
    ParamByName('EMP_NO').AsSmallInt := 52;
    ExecProc;
end;
```

Using output parameters

Stored procedures use output parameters to pass singleton data values to an application that calls the stored procedure. Output parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect output parameters from an application to retrieve its value after invoking the *TStoredProc.ExecProc* method.

Use the *TStoredProc.Params* property or *TStoredProc.ParamByName* method to reference the *TParam* object that represents a parameter and inspect its value. For example, to retrieve the value of a parameter and store it into the *Text* property of a *TEdit* component:

```
with StoredProc1 do begin
    ExecProc;
    Edit1.Text := Params[0].AsString;
end;
```

Most stored procedures return one or more output parameters. Output parameters may represent the sole return values for a stored procedure that does not also return a dataset, they may represent one set of values returned by a procedure that also returns a dataset, or they may represent values that have no direct correspondence to an individual record in the dataset returned by the stored procedure. Each server's implementation of stored procedures differs in this regard.

Note The source code for an Informix stored procedure may indicate that it returns output parameters even though you cannot not see output parameter information in the StoredProc Parameters editor. Informix translates output parameters into a single record dataset that you can view in your application's data-aware controls.

Using input/output parameters

Input/output parameters serve both function that input and output parameters serve individually. Applications use an input/output parameter to pass a singleton data value to a stored procedure, which in turn reuses the input/output parameter to pass a singleton data value to the calling application. As with input parameters, the input value for an input/output parameter must be set before the using stored procedure or query component is activated. Likewise, the output value in an input/output parameter will not be available until after the stored procedure has been executed.

In the example Oracle stored procedure below, the parameter `IN_OUTVAR` is an input/output parameter.

```
CREATE OR REPLACE PROCEDURE UPDATE_THE_TABLE (IN_OUTVAR IN OUT INTEGER)
AS
BEGIN
    UPDATE ALLTYPETABLE
    SET NUMBER82FLD = IN_OUTVAR
    WHERE KEYFIELD = 0;
    IN_OUTVAR:=1;
END UPDATE_THE_TABLE;
```

In the Delphi program code below, `IN_OUTVAR` is assigned an input value, the stored procedure executed, and then the output value in `IN_OUTVAR` is inspected and stored to a memory variable.

```
with StoredProc1 do begin
    ParamByName('IN_OUTVAR').AsInteger := 103;
    ExecProc;
    IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Using the result parameter

In addition to returning output parameters and a dataset, some stored procedures also return a single result parameter. The result parameter is usually used to indicate an error status or the number of records processed base on stored procedure execution. See your database server's documentation to determine if and how your server supports the result parameter. Result parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect a result parameter from an application to retrieve its value after invoking the `TStoredProc.ExecProc` method.

Use the `TStoredProc.Params` property or `TStoredProc.ParamByName` method to reference the `TParam` object that represents the result parameter and inspect its value.

```
DateVar := StoredProc1.ParamByName('MyOutputParam').AsDate;
```


Accessing parameters at design time

If you connect to a remote database server by setting the *DatabaseName* and *StoredProcName* properties at design time, then you can use the StoredProc Parameters editor to view the names and data types of each input parameter, and you can set the values for the input parameters to pass to the server when you execute the stored procedure.

Important Do not change the names or data types for input parameters reported by the server, or when you execute the stored procedure an exception is raised.

Some servers—Informix, for example—do not report parameter names or data types. In these cases, use the SQL Explorer or vendor-supplied server utilities to look at the source code of the stored procedure on the server to determine input parameters and data types. See the SQL Explorer online help for more information.

At design time, if you do not receive a parameter list from a stored procedure on a remote server (for example because you are not connected to a server), then you must invoke the StoredProc Parameters editor, list each required input parameter, and assign each a data type and a value. For more information about using the StoredProc Parameters editor to create parameters, see “Setting parameter information at design time” on page 22-13.

Setting parameter information at design time

You can invoke the StoredProc parameter collection editor at design time to set up parameters and their values.

The parameter collection editor allows you to set up stored procedure parameters. If you set the *DatabaseName* and *StoredProcName* properties of the *TStoredProc* component at design time, all existing parameters are listed in the collection editor. If you do not set both of these properties, no parameters are listed and you must add them manually. Additionally, some database types do not return all parameter information, like types. For these database systems, use the SQL Explorer utility to inspect the stored procedures, determine types, and then configure parameters through the collection editor and the Object Inspector. The steps to set up stored procedure parameters at design time are:

- 1 Optionally set the *DatabaseName* and *StoredProcName* properties.
- 2 In the Object Inspector, invoke the parameter collection editor by clicking on the ellipsis button in the *Params* field.
- 3 If the *DatabaseName* and *StoredProcName* properties are not set, no parameters appear in the collection editor. Manually add parameter definitions by right-clicking within the collection editor and selecting Add from the context menu.
- 4 Select parameters individually in the collection editor to display their properties in the Object Inspector.

- 5 If a type is not automatically specified for the *ParamType* property, select a parameter type (*Input*, *Output*, *Input/Output*, or *Result*) from the property's drop-down list.
- 6 If a data type is not automatically specified for the *DataType* property, select a data type from the property's drop-down list. (To return a result set from an Oracle stored procedure, set field type to *Cursor*.)
- 7 Use the *Value* property to optionally specify a starting value for an input or input/output parameter.

Right-clicking in the parameter collection editor invokes a context menu for operating on parameter definitions. Depending on whether any parameters are listed or selected, enabled options include: adding new parameters, deleting existing parameters, moving parameters up and down in the list, and selecting all listed parameters.

You can edit the definition for any *TParam* you add, but the attributes of the *TParam* objects you add must match the attributes of the parameters for the stored procedure on the server. To edit the *TParam* for a parameter, select it in the parameter collection editor and edit its property values in the Object Inspector.

Note Sybase, MS-SQL, and Informix do not return parameter type information. Use the SQL Explorer to determine this information.

Note Informix does not return data type information. Use the SQL Explorer to determine this information.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Creating parameters at runtime

If the name of the stored procedure is not specified in *StoredProcName* until runtime, no *TParam* objects will be automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method.

For example, the InterBase stored procedure GET_EMP_PROJ, below, requires one input parameter (EMP_NO) and one output parameter (PROJ_ID).

```
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 Delphi code to associate this stored procedure with a *TStoredProc* named *StoredProc1* and create *TParam* objects for the two parameters using the *TParam.Create* method is:

```

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;

```

Binding parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

Use the *ParamBindMode* property to specify how parameters in your stored procedure component 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. You can view a server's stored procedure source code in the SQL Explorer to determine the correct order and type of parameters to specify.

Viewing parameter information at design time

If you have access to a database server at design time, there are two ways to view information about the parameters used by a stored procedure:

- Invoke the SQL Explorer to view the source code for a stored procedure on a remote server. The source code includes parameter declarations that identify the data types and names for each parameter.
- Use the Object Inspector to view the property settings for individual *TParam* objects.

You can use the SQL Explorer to examine stored procedures on your database servers if you are using BDE native drivers. If you are using ODBC drivers you cannot examine stored procedures with the SQL Explorer. While using the SQL Explorer is not always an option, it can sometimes provide more information than the Object Inspector viewing *TParam* objects. The amount of information returned about a stored procedure in the Object Inspector depends on your database server.

To view individual parameter definitions in the Object Inspector:

- 1 Select the stored procedure component.
- 2 Set the *DatabaseName* property of a stored procedure component to the BDE alias for your database server (or the *DatabaseName* property of a *TDatabase*).
- 3 Set the *StoredProcName* property to the name of the stored procedure.
- 4 click the ellipsis button in for the *TStoredProc.Params* property in the Object Inspector.
- 5 Select individual parameters in the collection editor to view their property settings in the Object Inspector.

For some servers some or all parameter information may not be accessible.

In the Object Inspector, when viewing individual *TParam* objects, the *ParamType* property indicates whether the selected parameter is an input, output, input/output, or result parameter. The *DataType* property indicates the data type of the value the parameter contains, such as string, integer, or date. The *Value* edit box enables you to enter a value for a selected input parameter.

Note Sybase, MS-SQL, and Informix do not return parameter type information. Use the SQL Explorer or vendor-supplied server utilities to determine this information.

Note Informix does not return data type information. Use the SQL Explorer vendor-supplied server utilities to determine this information.

For more about setting parameter values, see “Setting parameter information at design time” on page 22-13.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

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.

Working with ADO components

The ADO components encapsulate the functionality of the ADO framework. ADO, or Microsoft ActiveX Data Objects, is a set of data objects that provide an application the ability to access data through an OLE DB provider. The Delphi ADO components encapsulate the functionality of these ADO objects and present their functionality in the context of Delphi components. The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are directly represented in the *TADOConnection*, *TADOCommand*, and ADO dataset components. There are other “helper” objects in the ADO framework, like the Field and Properties objects, but they are generally not used directly by the Delphi programmer and not represented by dedicated components.

Using ADO and the ADO components allows the Delphi programmer to create database applications that are not dependent on the Borland Database Engine (BDE), using instead ADO for the data access.

This chapter presents each of the ADO components and discusses how they differ from their BDE-based counterparts. References are given to topics covering aspects of the BDE-based connection and dataset components that are the same in the ADO equivalents.

Overview of ADO components

In addition to the connection and dataset components based on the Borland Database Engine (BDE), Delphi provides a set of components for use with ADO. These components allow the programmer to connect to an ADO data store and then to execute commands and retrieve data from tables in databases.

These ADO-centric data access components connect to ADO data stores and operate on data using only the ADO framework. The BDE is not employed at all in this process. Use the ADO components when ADO is available and you do not want to use the BDE. ADO 2.1 (or higher) must 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 of the ADO connection and dataset components are analogous to one of the BDE-based connection or dataset components. The *TADOConnection* component is functionally analogous to the *TDatabase* component in BDE-based applications. *TADOTable* is equivalent to *TTable*, *TADOQuery* to *TQuery*, and *TADOStoredProc* to *TStoredProc*. Use these ADO components in the same manner and context as you would the BDE-based data equivalents. *TADODataSet* has no direct BDE equivalent, but provides many of the same functions as *TTable* and *TQuery*. Similarly, there is no BDE component comparable to *TADOCommand*, which serves a specialized purpose in the Delphi/ADO environment.

The ADO components comprise the following classes.

Table 23.1 ADO components

Component	Use
<i>TADOConnection</i>	Used to establish a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and to operate on metadata.
<i>TADODataSet</i>	The primary component used to retrieve and operate on its data; can retrieve data from a single or multiple tables; can connect directly to a data store or through a <i>TADOConnection</i> .
<i>TADOTable</i>	Used to retrieve and operate on a dataset produced by a single table; can connect directly to a data store or through a <i>TADOConnection</i> .
<i>TADOQuery</i>	Used to retrieve and operate on a dataset produced by a valid SQL statement; can also execute data definition language (DDL) SQL statements, like CREATE TABLE; can connect directly to a data store or through a <i>TADOConnection</i> .
<i>TADOStoredProc</i>	Used to execute stored procedures; can execute stored procedures that retrieve data or execute DDL statements; can connect directly to a data store or through a <i>TADOConnection</i> .
<i>TADOCommand</i>	Used primarily to execute commands (SQL statements that do not return result sets); used with a supporting dataset component, can also retrieve a dataset from a table; can connect directly to a data store or through a <i>TADOConnection</i> .

Connecting to ADO data stores

Before commands can be executed or data retrieved, an application must establish a connection to a data store. While each individual ADO command and dataset component in an application can establish its own connection, a *TADOConnection* can be used and its single connection shared by other ADO components.

When connecting ADO command and dataset components to a data store, they can all use a shared connection or the components can each establish their own connections. Each approach has its own advantages and disadvantages.

This section covers the tasks involved in establishing and using a connection to an ADO data store.

Connecting to a data store using TADOConnection

One or more ADO dataset and command components can share a single connection to a data store. To do this, the application must have one *TADOConnection* component to make each data store connection. Then, the dataset and command components are associated with the connection component through their *Connection* properties.

In addition to providing the means for dataset and command components to connect to a data store, connection components provide properties and methods for activating and deactivating the connection, accessing the ADO connection object directly, and for determining what activity (if any) a connection component is engaged in at any given time.

Using a TADOConnection versus a dataset'sConnectionString

Each ADO command and dataset component in an application may be connected directly to a data store. However, when numerous command and dataset components are used, it is most often easier to maintain the connection using a single *TADOConnection* to establish the connection and then sharing that connection between the command and dataset components. See the section “Connecting to a data store using ADO dataset components” on page 23-13 for more information on connecting individual command and dataset directly to a data store.

Using a *TADOConnection* component to establish the connection offers more control over the connection versus connecting each command or dataset component individually. This greater control is provided by the properties, methods, and event of the *TADOConnection*, the functionality of which is not available otherwise.

Specifying the connection

To use a *TADOConnection* component to supply a shared connection for ADO dataset and command components, first establish the connection. Do this by supplying specific connection information in the *ConnectionString* property of the connection component. At design-time, invoke the connection string editor dialog by clicking the ellipsis button for the *ConnectionString* property in the Object Inspector. This dialog, supplied by the ADO system itself, allows you to interactively build a connection string by selecting connection elements (like the provider and server) from lists. At runtime, assign a **String** value to the *ConnectionString* property with the connection information. Setting the *Connected* property of the connection component to True would activate the connection. However, it is not essential to do so at this point. At design-time this is a good test of the connection, though.

```
ADOConnection1.ConnectionString := 'Provider=ProviderName;Remote Server=ServerReference';
```

The *ConnectionString* property can contain a number of connection parameters, each separated by semi-colons. These parameters can include the name of a provider, a user name and password (for login purposes), and a reference identifying a remote server. The *ConnectionString* property can also contain the name of a file containing the connection parameters. Such a file has the same contents as the *ConnectionString* property: one or more parameters, each with a value assignment and separated from

other parameters by a semi-colon. See the VCL help topic for the *ConnectionString* property for a list of ADO-supported parameters.

Once the connection information has been provided in the connection component, associate dataset and command components with the connection component. Do this by assigning a reference to the connection component to each dataset or command component's *Connection* property. 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 command below associates a *TADODataSet* component with a *TADOConnection* component.

```
ADODataSet1.Connection := ADOConnection1;
```

If you do not explicitly activate the connection by setting the connection component's *Connected* property to *True*, it will happen automatically when the first dataset component is activated or the first time a command is executed with a command component.

Accessing the connection object

Use the *ConnectionObject* property of *TADOConnection* to access the underlying ADO connection object directly. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object from an application.

Use of *ConnectionObject* to directly access the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in specific. It is not recommended that you use the Connection object directly unless familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

Activating and deactivating the connection

To activate an ADO connection component, set the *TADOConnection.Active* property to *True* or call the *TADOConnection.Open* method.

```
ADOConnection1.Active := True;
```

For the connection to be successful, the connection information provided in the *TADOConnection.ConnectionString* property must define a valid connection. For more information on providing connection information, see "Specifying the connection" on page 23-3.

Activating an ADO connection component will trigger the *OnWillConnect* and *OnConnectComplete* events of the ADO connection component and execute handlers for these events if they have been assigned.

If a connection component has not already been activated, it will automatically be activated if an associated dataset or command component is enabled. Dataset components cause this when they are activated. Command components do this when a command is executed. For information on associating dataset components with a

connection component, see “Connecting to a data store using TADOConnection” on page 23-3.

To deactivate an ADO connection component, either set its *Active* property to False or call its *Close* method.

```
ADOConnection1.Close;
```

Four things happen when a connection component is deactivated, using either the *Active* property or the *Close* method:

- 1 The *TADOConnection.OnDisconnect* event fires.
- 2 The handler for the *OnDisconnect* event executes (if one is assigned).
- 3 The *TADOConnection* component is deactivated.
- 4 Any associated ADO command or dataset components are deactivated.

Determining what a connection component is doing

At any time during the existence of a *TADOConnection* component, query its *State* property to determine what action, if any, in which the connection component is currently engaged.

A *TObjectStates* value of *stClosed* in the *TADOConnection.State* property indicates that the connection object is currently inactive. The *TADOConnection.Active* property contains a value of False and no associated command or dataset components are active.

A value of *stOpen* indicates that the connection component is active. A connection with an ADO data store has been successfully established, its *Active* property contains a value of True, and any one or more associated command or dataset components might be active.

A value of *stConnecting* indicates the connection component is currently attempting to establish a connection to the ADO data store specified in the *TADOConnection.ConnectionString* property. While still in this state, the *Cancel* method may be called to abort the connection attempt.

Fine-tuning a connection

When a *TADOConnection* component is used to make the connection to a data store for an application’s ADO command and dataset components, you have a greater degree of control over the conditions and attributes of the connection. These aspects are implemented using properties and event handlers of *TADOConnection* to fine-tune the connection.

Specifying connection attributes

Use the *TADOConnection.ConnectOptions* property to optionally force the connection to be asynchronous. By default, *ConnectOptions* 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*.

To set up a connection as asynchronous or to delegate the choice to the server, assign one of the *TConnectOption* constants to the connection component's *ConnectOptions* property. Then activate the connection component by calling its *Open* method, setting the *Connected* property to True, or by activating an associated command or dataset component. The example routines below respectively enable and disable asynchronous connections in the specified connection component.

```

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;

```

Use the *TADOConnection.Attributes* property to control the connection component's use of retaining commits and retaining aborts. *Attributes* can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. This makes controlling retaining commits and retaining aborts mutually exclusive using the same property.

Check whether either retaining commits or retaining aborts is enabled using the *in* operator with one of the constants. Enable one feature by adding the constant to the attributes property; disable one by subtracting the constant. The example routines below respectively enable and disable retaining commits in the specified connection component.

```

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;

```

Controlling timeouts

Control the period of time before attempted commands and connections are considered failed and are aborted using the *TADOConnection.ConnectionTimeout* property and the *TADOConnection.CommandTimeout* property.

ConnectionTimeout establishes the amount of time before an attempt to connect to the data store times-out. If the connection initiated by a call to the *Open* method has not successfully completed prior to expiration of the time specified in *ConnectionTimeout*, the connection attempt is cancelled. Set *ConnectionTimeout* to the number of seconds after which connection attempts time-out.

```
with ADOConnection1 do begin
  ConnectionTimeout = 10 {seconds};
  Open;
end;
```

CommandTimeout establishes the amount of time before attempted commands time-out. If the command initiated by a call to the *Execute* method has not successfully completed prior to expiration of the time specified in *CommandTimeout*, the command is cancelled and ADO generates an exception. Set *CommandTimeout* to the number of seconds after which commands time-out.

```
with ADOConnection1 do begin
  CommandTimeout = 10 {seconds};
  Execute('DROP TABLE Employeee1997', []);
end;
```

Controlling the connection login

An attempt to connect to a data store using a connection component triggers a security login event, *OnLogin*. One manifestation of this event is the appearance of a login dialog prompting for a user name and password. If desired, this dialog may be suppressed and the user name and password information supplied programmatically.

To suppress the default login dialog, first set the *LoginPrompt* property of the connection component to *False*. Then, prior to activating the connection component, supply all necessary login information via a vehicle such as the *ConnectionString* property.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
  ConnectionString := 'Provider=NameOfYourProvider;Remote Server=NameOfYourServer;' +
    'User Name=JaneDoe;Password=SecretWord';
  Connected := True;
end;
```

The login information can also be conveyed to the target data store as parameters for the connection component's *Open* method.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
```

```

ConnectionString := 'Provider=NameOfYourProvider;Remote Server=NameOfYourServer';
Open('JaneDoe', 'SecretWord');
end;

```

The second routine above is functionally equivalent to the first. The difference is that in the second routine the user name and password are not expressed in the *ConnectionString* property, but passed as parameters for the *Open* method. This is useful in situations where the connection specifications (like provider and server) are the same for all users and only the information particular to individual users changes. The application might, for instance, obtain the user-specific information via a custom login dialog, and the provider and server information form a static source like Windows Registry entries.

If, after supplying the login information programmatically, the login attempt is unsuccessful, an exception of type *EOleException* is raised.

Listing tables and stored procedures

The *TADODConnection* component provides properties for retrieving lists of the tables and stored procedures available through the connection. It also provides properties for accessing the dataset and command components associated with the connection component.

Accessing the connection's datasets

The *DataSets* and *DataSetCount* properties of *TADODConnection* allow a program to sequentially reference each dataset component associated with a connection component. Dataset components operated on by the *DataSets* and *DataSetCount* properties include *TADODDataSet*, *TADOQuery*, and *TADOStoredProc*. For working with a connection's command components, use the *Commands* and *CommandCount* properties.

DataSets is a zero-based array of references to ADO dataset components. Use an index with *DataSets* representing the position within the array of a particular dataset. For instance, use an index of 3 to reference the fourth dataset component in *DataSets*.

```
ShowMessage(ADODConnection1.DataSets[3].Name);
```

As *DataSets* provides a reference of type *TCustomADODDataSet*, typecast this reference as a descendent class type to access a property or call a method only available in a descendent class. For instance, *TCustomADODDataSet* does not have an *SQL* property, but the descendent class *TADOQuery* does. So, to access the *SQL* property of the dataset referenced through the *DataSets* property, typecast it as a *TADOQuery*.

```

with (ADODConnection1.DataSets[10] as TADOQuery do begin
  SQL.Clear;
  SQL.Add('SELECT * FROM Species');
  Open;
end;

```

The *DataSetCount* property provides a total count of all of the datasets associated with a connection component. You can use the *DataSetCount* property as the basis for a loop with the *DataSets* property to sequentially visit all of the dataset components associated with a connection.

```

var
  i: Integer
begin
  for i := 0 to (ADOConnection4.DataSetCount) do
    ADOConnection4.DataSets[i].Open;
  end;
end;

```

Accessing the connection's commands

The *Commands* and *CommandCount* properties of *TADOConnection* act in much the same manner as the *DataSets* and *DataSetCount* properties. The difference is that *Commands* and *CommandCount* provide references to all of the *TADOCommand* components associated with the connection component. To work with all of the connection's dataset components, use the *DataSets* and *DataSetCount* properties.

Commands is a zero-based array of references to ADO command components. Use an index with *Commands* representing the position within the array of a particular command. For instance, use an index of 1 to reference the second command component in *Commands*.

```

Memo1.Lines.Text := ADOConnection1.Commands[1].CommandText;

```

The *CommandCount* property provides a total count of all of the commands associated with a connection component. You can use the *CommandCount* property as the basis for a loop with the *Commands* property to sequentially visit all of the command components associated with a connection.

```

var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount) do
    ADOConnection1.Commands[i].Execute;
  end;
end;

```

Listing available tables

To get a listing of all of the tables contained in the database accessed via the connection object, use the *GetTableNames* method. This method copies a list of table names to an already-existing string list object. Use individual elements from this list for such things as the value for the *TableName* property of a *TADOTable* component or the name of a table in an SQL statement executed by a *TADOQuery*.

```

ADOConnection1.GetTableNames(ListBox1.Items, False);

```

The example below traverse a list of table names created using the *GetTableNames* method. For each table, the routine makes an entry in another table with the table's name and number of records.

```

procedure TForm1.Button1Click(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try

```

```

ADOConnection1.GetTableNames(SL, False);
for index := 0 to (SL.Count - 1) do begin
  Table1.Insert;
  Table1.FieldByName('Name').AsString := SL[index];
  ADOTable1.TableName := SL[index];
  ADOTable1.Open;
  Table1.FieldByName('Records').AsInteger :=
    ADOTable1.RecordCount;
  Table1.Post;
end;
finally
  SL.Free;
  ADOTable1.Close;
end;
end;

```

Listing available stored procedures

To get a listing of all of the stored procedures contained in the database accessed via the connection object, use the *GetProcedureNames* method. This method copies a list of stored procedure names to an already-existing string list object. One of the elements in the resulting list can be used for such things as the value for the *ProcedureName* property of a *TADOStoredProc* component.

```
ADOConnection1.GetProcedureNames(ListBox1.Items);
```

In the example below, a list of stored procedure names retrieved with *GetProcedureNames* is used to execute all of the stored procedures from the associated database.

```

procedure TDataForm.ExecuteProcsButtonClick(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try
    ADOConnection1.GetProcedureNames(SL);
    if (SL.Count > 0) then
      for index := 0 to (SL.Count - 1) do begin
        ADOStoredProc1.ProcedureName := SL[index];
        ADOStoredProc1.ExecProc;
      end;
    finally
      SL.Free;
    end;
  end;

```


Working with (connection) transactions

The *TADOConnection* component includes a number of methods and events for working with transactions. These transaction capabilities are shared by all of the ADO command and dataset components using the data store connection.

Using transaction methods

Use the methods *BeginTrans*, *CommitTrans*, and *RollbackTrans* to perform transaction processing. *BeginTrans* starts a transaction in the data store associated with the ADO connection component. *CommitTrans* commits a currently active transaction, saving changes to the database and ending the transaction. *RollbackTrans* cancels a currently active transaction, abandoning all changes made during the transaction and ending the transaction. Read the *InTransaction* property to determine at any given point whether the connection component has a transaction open.

A transaction started by the connection component is shared by all command and dataset components that use the connection established by the *TADOConnection* component.

Using transaction events

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 have been successfully completed at the data store.

The *OnBeginTransComplete* event is triggered when the data store has successfully started a transaction after a call to the connection component's *BeginTrans* method. The *OnCommitTransComplete* event is triggered after a transaction is successfully committed due to a call to *CommitTrans*. And *OnRollbackTransComplete* is triggered after a transaction is successfully committed due to a call to *RollbackTrans*.

Using ADO datasets

The ADO dataset components provided in Delphi are analogous to the BDE-based dataset components. For instance, the *TADOTable* component is functionally equivalent to the *TTable* component. The main difference is that the ADO dataset components use underlying ADO objects for their data access and are not dependent on the Borland Database Engine for this.

The ADO dataset and BDE-based components have the *TDataSet* class as a common ancestor. Because of this, they share a common functionality in inherited or similar properties, methods, and events. This section primarily discusses areas of the ADO dataset components that differ from the corresponding generic dataset components. For more information on functionality common between the two sets of dataset components, see the descriptions for the generic and BDE-based dataset components:

- “Understanding datasets” on page 18-1

- “Working with tables” on page 20-1
- “Working with queries” on page 21-1
- “Working with stored procedures” on page 22-1

This section contains information pertaining to functionality that differs in the ADO versions of the dataset components from their generic counterparts. This information is divided into the areas:

- Features common to all ADO dataset components
- Using TADODataset
- Using TADOTable
- Using TADOQuery
- Using TADOStoredProc

Features common to all ADO dataset components

Certain aspects of the ADO dataset components function exactly the same in all of the different components. Except as cited, these functional areas are used in exactly the same manner no matter which ADO dataset component is in use.

Modifying data

Accessing columns in ADO dataset components and modifying data is done in the exact same manner as in the generic dataset components. Use dataset methods like *Edit* and *Insert* to put the dataset in edit mode prior to changing data. Use the *Post* method to finalize data changes.

Use the dynamic *TField* references provided by the *Fields* property and *FieldByname* method of the dataset components. From there use the properties and methods of the *TField* class and descendents to do such things as setting or getting a column’s value, validating data, and determining a column’s data type.

For information on modifying data through dataset components, see “Modifying data” on page 18-20. For information on using table columns and persistent field objects, see “Working with field components” on page 19-1.

Navigating in a dataset

Navigating between rows in an ADO dataset is done in the same way as in generic dataset components. Use methods like *First*, *Next*, *Last*, and *Prior* to move the record pointer in the dataset component from one table row to another. Loops can be based on the *Eof* and *Bof* properties so that they operate on all of the rows that make up a dataset.

```
ADOTable1.First;
while not ADOTable1.Eof do begin
  { Process each record here }
  :
  ADOTable1.Next;
end;
```

For information on navigating between table rows in dataset components, see “Navigating datasets” on page 18-9.

Using visual data-aware controls

The dataset provided by an ADO dataset component can be made available in an application using data-aware controls. Such datasets include the rows returned by a *TADOTable* component, the result set returned by a SELECT statement in a *TADOQuery*, and stored procedures that return a result set executed from a *TADOStoredProc* component.

To make these datasets available in data-aware controls:

- 1 Use a standard *TDataSource* component.
- 2 Specify an active ADO dataset component in its *DataSet* property.
- 3 Use the standard data-aware controls, like *TDBEdit* and *TDBGrid*.
- 4 Specify the *TDataSource* in the *DataSource* property of the data-aware control.

For example, creating this relationship between ADO dataset component, datasource component, and data-aware control programmatically:

```
DBGrid1.DataSource := DataSource1;
DataSource1.DataSet := ADOQuery1;
ADOQuery1.Open;
```

Connecting to a data store using ADO dataset components

ADO dataset components can connect to an ADO data store either collectively or individually.

When connecting dataset components collectively, set the *Connection* property of each dataset component to a *TADOConnection* component. Each dataset component then uses the connection established by that connection component.

```
ADODataset1.Connection := ADOConnection1;
ADODataset2.Connection := ADOConnection1;
...
```

Among the advantages of connecting dataset components collectively are:

- The dataset components share the connection object’s attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The dataset components can participate in transactions.

When connecting dataset components individually, set the *ConnectionString* property of each dataset component. The information needed to connect to the data store must be set for each dataset component. Each dataset component establishes its own connection to the data store, totally independent of any other dataset connection in the application.

```
ADODataset1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
ADODataset2.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
...
```

For more information on using a TADOConnection to connect to a data store see “Connecting to a data store using TADOConnection” on page 23-3.

Working with record sets

In addition to the means for navigating between records and modifying data shared by all ADO dataset components, there are a number of other properties and methods for operating on 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. It is not recommended that you use the recordset object directly unless familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

Use the *RecordSetState* property to determine the current state of the dataset component. *RecordSetState* implements the *State* property of the ADO recordset object, and so reflects the current state of the underlying recordset object. The *RecordSetState* property will contain one of the values: *stExecuting* or *stFetching*. A value of *stExecuting* indicates the dataset component is currently in the process of executing a command. A value of *stFetching* indicates the dataset component is in the process of fetching rows from the associated table (or tables).

Use these 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.

Using batch updates

ADO dataset components provide the ability to cache changes to the dataset and then either apply all of the changes as a batch operation or to cancel one or all of the changes. Batch updates can serve as a sort of transaction control, but at the dataset component level. (Ordinarily, transactions are handled as methods of the ADO connection component.)

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 to the values 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.

```
with ADODataset1 do begin
  CursorLocation := clUseServer;
  CursorType := ctKeyset;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

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.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:   StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:       StatusBar1.Panels[0].Text := 'New record';
end;
```

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.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

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 *TUpdateBatchOptions* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *ubAffectAll* is passed, only a subset of the pending changes are applied. Passing *ubAffectAll* 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:

```
ADODataset1.UpdateBatch(ubAffectCurrent);
```

Canceling batch updates

Cancel pending data changes that have not yet been canceled or applied by calling the *CancelBatch* method. Rows that have been changed and are canceled have their columns values reverted back 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 *TUpdateBatchOptions* value can optionally be passed as the parameter for *CancelBatch*. If any value except *ubAffectAll* is passed, only a subset of the pending changes are canceled. Passing *ubAffectAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataset1.Cancel;
```

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. Save the data to a file using the *SaveToFile* method. Retrieve the data from file using the *LoadFromFile* method. The data is saved in one of two proprietary formats: ADTG and XML. Indicate which of these two formats to use for the save file with one of the *TPersistFormat* constants *pfADTG* or *pfXML* in the *Format* parameter of the *SaveToFile* method.

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*.

```
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;

```

The saving and loading dataset components 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.

On calling the *LoadFromFile* method, the dataset component is automatically activated.

If the file specified in the *FileName* parameter of the *SaveToFile* method already exists, an *EOleException* exception is raised. Similarly, if the file specified in the *FileName* parameter of *LoadFromFile* does not exist, an *EOleException* exception is raised.

The two save file formats ADTG and XML are the only formats supported by ADO. Even so, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the actual version in use to determine what save file formats are supported.

Using parameters in commands

Using parameters in commands and SQL statements executed as commands using ADO dataset components requires that you:

- 1 Include parameters in the SQL statement (identified by the prefixing colon).
- 2 Set the property values for each *TParameter* component.

For each token in the SQL statement identified as a parameter, one *TParameter* component is automatically created and added to the dataset component's *Parameters* property (a *TParameters* array of *TParameter* components). At design-time, access the parameter components to set their values using the property editor for the *Parameters* property. To invoke the property editor, click the ellipsis button for the *Parameters* property in the Object Inspector.

At runtime, access parameter components to set or get their values using the *Parameters* property of the dataset component. Specify an index number with *Parameters* that is the ordinal position of a specific parameter in the SQL statement (relative to other parameters). This index is zero based, so the first parameter is referenced with an index of zero, the second with an index of one, and so on. Alternately, use the *TParameters* reference provided by the *Parameters* property and call its *ParamByName* method to refer to the parameter by its name.

```

{ reference the first parameter with an index }
ADOQuery1.Parameters[0].Value := 'telephone';

{ reference a parameter by its name }
ADOQuery1.Parameters.ParamByName('Amount').Value := 123;

```

In the example below, the following SQL statement is used in a *TADOQuery* component.

```
SELECT CustNo, Company, State
FROM CUSTOMER
WHERE (State = :StateParam)
```

This statement has one parameter: *StateParam*. The routine below closes the ADO query component, sets the value of the *StateParam* parameter through the *Parameters* property, and then reopens the ADO query component. The *Parameters* property requires a parameter be identified by a number representing the parameter's ordinal position in the statement, relative to other parameters. *Parameters* is zero-based, so the first parameter is identified with a *Parameters* property index of zero, the second with a one, and so on. As *StateParam* is the first parameter in the statement, an index of zero is used to identify it.

```
procedure TForm1.GetCaliforniaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters[0].Value := 'CA';
    Open;
  end;
end;
```

The procedure below performs essentially the same purpose, but uses the *TParameters.ParamByName* method to set the parameter's value. The *ParamByName* method requires a parameter be identified by its name as used in the SQL statement (sans the colon).

```
procedure TForm1.GetFloridaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters.ParamByName('StateParam').Value := 'FL';
    Open;
  end;
end;
```

Using TADODataset

The *TADODataset* component provides Delphi applications the ability to access data from one or multiple tables in a database accessed via ADO. Tables accessed are specified using the *CommandText* property of the ADO dataset component, either by name or using an SQL statement.

The database is accessed using a data store connection established by the ADO dataset component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property. See "Connecting to a data store using ADO dataset components" on page 23-13 for more information on this.

Using data provided by a *TADODataset* component in visual controls, navigating through the rows, and programmatically modifying the data is the same as for the rest of the ADO dataset components. See “Features common to all ADO dataset components” on page 23-12 for more information on features common to all dataset components.

Retrieving a dataset using a command

The *TADODataset* component is capable of retrieving data from a single table using the name of a table. It can also retrieve data from one or multiple tables using a valid SQL statement. In either case, the table name or SQL statement is executed as a command.

Specify the name of a table or an SQL statement in the *CommandText* property and activate the component. At design-time, you can use the Command Text Editor to build the command. To invoke this editor, click the ellipsis button in the *CommandText* property in the Object Inspector. At runtime, assign a command to *CommandText* as a **String**.

```
ADODataset1.CommandText := 'SELECT * FROM Customer';
```

Use the *CommandType* property to indicate the type of command being executed: *cmdTable* (or *cmdTableDirect*) if the command is a table name or *cmdText* for SQL statements. You can also specify *cmdUnknown* if the command type is not known at time or execution or you wish ADO to make a guess at the command type based on the contents of *CommandText*. At design-time, select the desired value for *CommandType* from the drop-down list in the Object Inspector. At runtime, assign a value of type *TCommandType*.

```
ADODataset1.CommandType := cmdText;
```

Activate the *TADODataset* by calling its *Open* method or by assigning a value of True to the *Active* property.

```
with ADODataset1 do begin
  Connection := ADOConnection1;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Customer';
  Open;
end;
```

Using TADOTable

The *TADOTable* component provides Delphi applications the ability to access data from a single table in a database accessed via ADO. The table accessed is specified in the *TableName* property of the ADO table component.

The database is accessed using a data store connection established by the ADO table component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property. See “Connecting to a data store using ADO dataset components” on page 23-13 for more information on this.

Using data provided by a *TADOTable* component in visual controls, navigating through the rows, and programmatically modifying the data is the same as for the rest of the ADO dataset components. See “Features common to all ADO dataset components” on page 23-12 for more information on features common to all dataset components.

Specifying the table to use

Once a *TADOTable* component has a valid connection to a database, it can access tables contained in that database. Specify a single table of the database in the *TableName* property. When the ADO table component is activated, the table and its data become accessible through the *TADOTable*.

At design-time, if the *TADOTable* component has a valid data store connection, the property editor for the *TableName* property lists the names of available tables. Select one table from this list. At runtime, assign a **String** value containing a table name to the *TableName* property.

```
ADOTable1.TableName := 'Orders';
```

If a *TADOConnection* component is used to connect to the data store, you can use its *GetTableNames* method to retrieve a list of available tables. *GetTableNames* fills an already-existing string list object with the names of the tables available through the connection.

For example, the first routine below fills a *TListBox* component named *ListBox1* with the names of tables available through the *TADOConnection* component *ADOConnection1*. The second routine is a handler for the *OnDbClick* event of *ListBox1*. In this event handler, the currently selected table name in *ListBox1* is assigned to the *TableName* property of the *TADOTable* called *ADOTable1*. The ADO table component is then activated.

```
procedure TForm1.ListTablesButtonClick(Sender: TObject);
begin
  ADOConnection1.GetTableNames(ListBox1.Items, False);
end;

procedure TForm1.ListBox1DbClick(Sender: TObject);
begin
  with ADOTable1 do begin
    Close;
    TableName := (Sender as TListBox).Items[(Sender as TListBox).ItemIndex];
    Open;
  end;
end;
```

Using TADOQuery

The *TADOQuery* component provides Delphi applications the ability to access data from one or multiple tables from an ADO database using SQL. Specify the SQL statement to use with the ADO query component in the *SQL* property. *TADOQuery* can either retrieve data using data manipulation language (DML) or create and delete metadata objects using data definition language (DDL). The SQL used in a

TADOQuery component must be acceptable to the ADO driver in use. Delphi performs no evaluation of the SQL and does not execute it. The SQL statement is merely passed to the database back-end for execution. If the SQL statement produces a result set, it is passed from the database back-end through Delphi to the *TADOQuery* for use by the application.

The database is accessed using a data store connection established by the ADO query component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property. See “Connecting to a data store using ADO dataset components” on page 23-13 for more information on this.

Using data provided by a *TADOQuery* component in visual controls, navigating through the rows, and programmatically modifying the data is the same as for the rest of the ADO dataset components. See “Features common to all ADO dataset components” on page 23-12 for more information on features common to all dataset components.

Specifying SQL statements

At design-time, invoke the property editor for the *SQL* property by clicking the ellipsis button in the Object Inspector. In the editor dialog, enter the SQL statement for the *TADOQuery*.

At runtime, assign a value to the *SQL* property. As is the case with the standard querying component, *TQuery*, the *TADOQuery.SQL* property is a string list object. Use properties and methods of the string list class to assign values to the *SQL* property.

In the example below, a *SELECT* statement is assigned to the *SQL* property of a *TADOQuery* component named *ADOQuery1*.

```
with ADOQuery1 do begin
  Close;
  with SQL do begin
    Clear;
    Add('SELECT Company, State');
    Add('FROM CUSTOMER');
    Add('WHERE State = ' + QuotedStr('HI'));
    Add('ORDER BY Company');
  end;
  Open;
end;
```

Executing SQL statements

A *TADOQuery* with a valid SQL statement in its *SQL* property can be executed in one of two ways. Which way you use is predicated on the type of SQL statement the ADO query component is to execute.

If the SQL statement is one that returns a result set, the ADO query component should be activated by calling its *Open* method or by settings its *Active* property to

True. Only SELECT statements return result sets, so a *TADOQuery* with a SELECT statement in its SQL property will always be activated using this approach.

```
ADOQuery1.SQL.Text := 'SELECT * FROM TrafficViolations';
ADOQuery1.Open;
```

Note that because methods cannot be called while designing an application in the Delphi IDE, only the *Active* property can be used to activate this kind of query at design-time. This is functionally the same as calling the *Open* method (at runtime).

Execute an SQL statement that does not return a result set by calling the *ExecSQL* method of the *TADOQuery* component. All SQL statements except SELECT fall into this category: INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE, and so on. A *TADOQuery* component with one of these SQL statements in its SQL property will always be activated using this approach.

```
ADOQuery1.SQL.Text := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
ADOQuery1.ExecSQL;
```

The *TADOCommand* component can be used to execute SQL statements like the one above that do not return result sets.

Using TADOStoredProc

The *TADOStoredProc* component provides Delphi applications the ability to execute stored procedures in a database accessed through an ADO data store. The stored procedure executed is specified in the *ProcedureName* property of the ADO stored procedure component.

The database is accessed using a data store connection established by the stored procedure component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property. See “Connecting to a data store using ADO dataset components” on page 23-13 for more information on this.

Result sets retrieved by a *TADOStoredProc* component are made available to an application in the same manner as the standard, BDE-centric query component *TStoredProc*. Use the ADO stored procedure component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO stored procedure component and data-aware controls. See “Using visual data-aware controls” on page 23-13 for more information on this.

Using data provided by a *TADOStoredProc* component in visual controls, navigating through the rows, and programmatically modifying the data is the same as for the rest of the ADO dataset components. See “Features common to all ADO dataset components” on page 23-12 for more information on features common to all dataset components.

Specifying the stored procedure

Once a *TADOStoredProc* component has a valid connection to a database, it can execute stored procedures contained in that database. Specify the name of a stored procedure from the database in the *ProcedureName* property. Activate the ADO

stored procedure component using its *Open* method (if it returns a result set) or its *ExecProc* method (if it does not).

At design-time, if the *TADOStoredProc* component has a valid data store connection, the property editor for the *ProcedureName* property lists the names of available stored procedures. Select a stored procedure from this list. At runtime, assign a **String** value containing a stored procedure name to the *ProcedureName* property.

```
ADOStoredProc1.ProcedureName := 'DeleteEmployee';
```

If a *TADOConnection* component is used to connect to the data store, you can use its *GetProcedureNames* method to retrieve a list of available stored procedures. *GetProcedureNames* fills an already-existing string list object with the names of the stored procedures available through the connection.

For example, the first routine below fills a *TListBox* component named *ListBox1* with the names of stored procedures available through the *TADOConnection* component *ADOConnection1*. The second routine is a handler for the *OnDbClick* event of *ListBox1*. In this event handler, the currently selected table name in *ListBox1* is assigned to the *ProcedureName* property of the *TADOStoredProc* called *ADOStoredProc1*. The ADO stored procedure component is then executed using the *ExecProc* method.

```
procedure TForm1.ListProceduresButtonClick(Sender: TObject);
begin
  ADOConnection1.GetProcedureNames(ListBox1.Items);
end;

procedure TForm1.ListBox1DbClick(Sender: TObject);
begin
  with ADOStoredProc1 do begin
    ProcedureName := TListBox(Sender).Items[TListBox(Sender).ItemIndex];
    ExecProc;
  end;
end;
```

Executing the stored procedure

A *TADOStoredProc* with the name of an existing stored procedure in its *ProcedureName* property can be executed in one of two ways. Which way is you use is predicated on whether or not the stored procedure returns a result set.

If the stored procedure is one that returns a result set, the ADO stored procedure component should be activated by calling its *Open* method or by settings its *Active* property to True.

```
ADOStoredProc1.ProcedureName := 'ShowPurebreds';
ADOStoredProc1.ExecProc;
```

Note that as methods cannot be called while designing an application in the Delphi IDE, only the *Active* property can be used to activate this kind of stored procedure at design-time.

Execute a stored procedure that does not return a result set by calling the *ExecProc* method of the *TADOStoredProc* component. All SQL statements except SELECT fall into this category: INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE,

and so on. A *TADOQuery* component with one of these SQL statements in its *SQL* property will always be activated using this approach.

```
ADOStoredProc1.ProcedureName := 'DeletePoodles';
ADOStoredProc1.ExecProc;
```

Using parameters with stored procedures

The *TADOStoredProc* component is capable of accommodating three types of parameters (not all of which may be supported by all database types). A parameter may be for input, for output, or for returning a result set. This section describes using stored procedure parameters in these three roles. It is possible for a parameter to serve two purposes, such as being both an input and an output parameter. This is merely a variation on the three basic roles. Dual use of a parameter like this is a matter of combining two of the functional approaches described here.

The direction or purpose of a parameter is defined in the stored procedure when it is created. This direction cannot later be changed by a front-end application. For instance, you cannot use Delphi code to turn an input parameter into an output parameter. The stored procedure would need to be dropped and recreated, giving the parameter a new role in the process. The direction of a particular parameter is indicated in the *TParameter.Direction* property, which can be read either at design-time in the Object Inspector or programmatically at runtime.

Table 23.2 Parameter direction property

Parameter Direction	Use
<i>pdInput</i>	Parameter used to supply a value to the stored procedure before execution.
<i>pdOutput</i>	Parameter used to return a singleton value from a stored procedure after execution.
<i>pdInputOutput</i>	Parameter that can be used as both an input and an output parameter, per the above definitions.
<i>pdReturnValue</i>	Parameter that contains a result set after execution.
<i>pdUnknown</i>	Parameter for which the direction could not be determined at the point of evaluation.

Using TADOStoredProc input parameters

Use an input parameter to supply a value to a stored procedure before executing that stored procedure. Such values are typically used in the WHERE clause of a stored procedure's SQL statement to limit the number of table rows affected. Assign the parameter a value before activating the *TADOStoredProc* by calling its *Open* method or setting its *Active* property to True (for stored procedures that return a result set) or before executing the *TADOStoredProc* with its *ExecProc* method.

At design-time, access parameters through the Object Inspector. With focus in the cell for the *Parameters* property, click the ellipsis button. This invokes the parameters editor dialog. Enter a value of the appropriate type in the *Value* property.

At runtime, assign a value to the *Value* property of the *TParameter* component for the target parameter. Use the *TParameter* reference provided by the *Parameters* property of the *TADOStoredProc*.

```
with ADOStoredProc1 do begin
  Close;
  Parameters[1].Value := 1;
  Open;
end;
```

The *TParameters*::*Param*

Using TADOStoredProc output parameters

Use an input parameter to return a single value from a stored procedure. While a result set might consist of multiple rows of multiple columns, this output parameter must be the equivalent of one row containing one column. If the stored procedure uses a *SELECT* statement to retrieve this value, an attempt to return multiple values results in an exception.

An output parameter only contains a value after the stored procedure has been activated or executed. Note that not all database systems support returning both a result set and output parameters. Check the documentation for the particular database system you are using to verify what it supports in this regard. If a database system supports both returning a result set and output parameter values, the *TADOStoredProc* can be activated using either its *Open* method (or *Active* property) or its *ExecProc*. If the database system does not support both, you can only use output parameters to retrieve values by calling the *ExecProc* method.

At design-time, access parameters through the Object Inspector. With focus in the cell for the *Parameters* property, click the ellipsis button. This invokes the parameters editor dialog. If the *TADOStoredProc* is activated using its *Active* property (the only way to activate it at design-time), inspect the output parameter's *Value* property to see the value returned.

At runtime, assign a value to the *Value* property of the *TParameter* component for the target parameter. Use the *TParameter* reference provided by the *Parameters* property of the *TADOStoredProc*. For example, the routine below returns a **String** value through the output parameter named *@OutParam1*.

```
with ADOStoredProc1 do begin
  Close;
  ShowMessage(VarToStr(Parameters.ParamByName('@OutParam1').Value));
  ExecProc;
end;
```

Using TADOStoredProc return value parameters

Return value parameters need not be accessed directly. Instead, the result set returned through a return value parameter should be accessed as you would any dataset.

To make the result set available through visual data-aware controls, use a reference to the *TADOStoredProc* component as the value for the *DataSet* property of a

TDataSource component. The *TDataSource* then acts as a conduit between the *TADOStoredProc* component and the data-aware controls. At design-time, this is done through the Object Inspector. In the *DataSet* property of the *TDataSource*, select the *TADOStoredProc* component from the drop-down list. At runtime, assign a reference to the *TADOStoredProc* to the *DataSet* property.

```
ADOStoredProc1.Close;
DataSource1.DataSet := ADOStoredProc1;
ADOStoredProc1.Open;
```

Alternately, the result set can be accessed and manipulated using navigation and editing properties and methods inherited from *TDataSet*. For information on modifying data through dataset components, see “Modifying data” on page 18-20. For information on navigating between table rows in dataset components, see “Navigating datasets” on page 18-9.

Executing commands

The set of ADO components provided in Delphi allows an application to execute commands. This section describes how to execute commands and what components to use to do so.

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.

Commands can be executed from more than one Delphi component. Each command-capable component executes commands in a slightly different way, with varying strengths and weaknesses. Which component you should use for a particular command is predicated on the type of command and whether it returns a result set. In general, for commands that do not return a result set use the *TADOCommand* component (though the *TADOQuery* component can also execute these commands). For commands that do return a result set, either execute the command from a *TADODataSet* or use the command’s statement in the *SQL* property of a *TADOQuery*.

The *TADOCommand* component provides the ability to execute commands, one command at a time. It is designed 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, though, it is capable of returning a result set that can be used through an ADO dataset component.

Specify commands in the *CommandText* property. A command can optionally be described using the *CommandType* property. If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*. Commands used with an ADO command component can contain parameters for which values are substituted before execution of the command. Before a command can be executed, the ADO command component must have a valid connection to an ADO data store.

Specifying the command

Specify the command to execute using the ADO command component in its `CommandText` property. At design-time, enter the command (an SQL statement, a table name, or the name of a procedure) in the `CommandText` property through the Object Inspector. At runtime, assign a **String** value containing the command to the `CommandText` property.

If desired, explicitly define the type of command being executed in the `CommandType` property. Among the constants for `CommandType` are: `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.

```
with ADOCommand1 do begin
    CommandText := 'AddEmployee';
    CommandType := cmdStoredProc;
    ...
end;
```

Using the Execute method

Before the command can be executed using an ADO command component, the `TADOCommand` must have a valid connection to a data store. See “Connecting to a data store using ADO dataset components” on page 23-13 for more information this.

To execute the command call the `Execute` method of the ADO command component. For commands that do not require any parameters or execution options, call the simple overloaded version of `Execute` without any method parameters at all.

```
with ADOCommand1 do begin
    CommandText := 'UpdateInventory';
    CommandType := cmdStoredProc;
    Execute;
end;
```

For information on executing commands that return a result set, see “Retrieving result sets with commands” on page 23-28.

Canceling commands

After an attempt to execute a command has been initiated (with the `Execute` method of a `TADOCommand` component), it can be aborted by calling the `Cancel` method.

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
    ADOCommand1.Execute;
end;

procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
    ADOCommand1.Cancel;
end;
```

The *Cancel* method only has an effect if there is a command pending and the command 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 and the command has not yet been completed or timed out. If a command has not been aborted or completed before the number of seconds specified in the *CommandTimeout* property have expired, the command times out. If a timeout period of other than the default 30 seconds is desired, set the *CommandTimeout* property prior to calling the *Execute* method.

Retrieving result sets with commands

Executing a command that returns a result set is exactly the same as for those that do not, except that a pre-existing ADO dataset component must represent the result set. The *Execute* method of *TADOCCommand* returns an ADO recordset object. Assign this return value to the *RecordSet* property of an ADO dataset component such as a *TADODataset*.

In the example below, the ADO record set produced by a call to the *Execute* method of a *TADOCCommand* component (*ADOCCommand1*) is assigned to the *Recordset* property of a *TADODataset* component (*ADODataset1*).

```
with ADOCCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataset1.Recordset := Execute;
end;
```

As soon as this assignment is made to the ADO dataset component's *Recordset* property, the dataset component is activated (automatically) and the data is available. Use methods and properties of the dataset component to access the data programmatically. To make the data available using visual data-aware controls, use a *TDataSource* component as a conduit between the ADO dataset and the data-aware controls.

For information on executing commands that do not return a result set, see "Executing commands" on page 23-26.

Handling command parameters

Executing a command that has parameters is exactly the same as for those that do not, except that values must be assigned to the parameters before the command is executed.

For each parameter in the command, one *TParameter* object is automatically added to the *Parameters* property of the *TADOCCommand* component. At design-time use the Parameter Editor to access parameters, which is invoked 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.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Access single *TParameter* objects in *Parameters* by a number representing the relative position (to each other) in the command using the *Parameters* property of the *TADOCommand* component. Reference the *TParameter* objects by their names using the *TParameters.ParamByName* method.

Creating and using a client dataset

TClientDataSet is a dataset component designed to work without the connectivity support of the Borland Database Engine (BDE) or ActiveX Data Objects (ADO). Instead, it uses *MIDAS.DLL*, which is much smaller and simpler to install and configure. You don't use database or ADO connection components with client datasets, because there is no database connection.

Client datasets provide all the data access, editing, navigation, data constraint, and filtering support introduced by *TDataSet*. However, the application that uses a client dataset must provide the mechanism by which the client dataset reads data and writes updates. Client datasets provide for this in one of the following ways:

- Reading from and writing to a flat file accessed directly from a client dataset component. This is the mechanism used by flat-file database applications. For more information about using a client dataset in flat-file applications, see "Flat-file database applications" on page 13-13.
- Reading from another dataset. Client datasets provide a variety of mechanisms for copying data from other datasets. These are described in "Copying data from another dataset" on page 24-12.
- Using an *IAppServer* interface to obtain data from and post updates to a remote application server. This is the mechanism used by clients in a multi-tiered database application. For more information about building multi-tiered database applications, see Chapter 14, "Creating multi-tiered applications".

These mechanisms can be combined into a single application that employs the "briefcase model". Users take a snapshot of data, saving it to a flat-file so that they can work on it off-line. Later, the client application applies the changes from the local copy of data to the application server. The application server resolves them with the actual database, and returns errors to the client dataset for handling. For information building an application using the briefcase model, see "Using the briefcase model" on page 13-17.

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 Chapter 26, “Using data controls” for information on how to display database information in data-aware controls.

Because *TClientDataSet* is a descendant of *TDataSet*, client datasets inherit the power and usefulness of the properties, methods, and events defined for all dataset components. For a complete introduction to this generic dataset behavior, see Chapter 18, “Understanding datasets.”

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for common database functions can involve additional capabilities or considerations.

Navigating data in client datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset’s records just as for any other dataset. You can also navigate programmatically through records using standard dataset methods such as *First*, *GotoKey*, *Last*, *Next*, and *Prior*. For more information about these methods, see “Navigating datasets” on page 18-9.

Client datasets also support standard bookmark capabilities for marking and navigating to specific records. For more information about bookmarking, see “Marking and returning to records” on page 18-12.

Unlike most datasets, client datasets can also position the cursor at any 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” on page 18-16. For more information about ranges, see “Working with a subset of data” on page 20-11.

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 with 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. See the online documentation for *TClientSet.Filter* for details.

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. In multi-tiered applications, you can also limit the data that is stored in the client dataset to a subset of records by supplying parameters to the application server. For more information on this, see “Limiting records with parameters” on page 24-16.

Representing master/detail relationships

Like tables, client datasets support master/detail forms. 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). For more information about master/detail forms, see “Creating master/detail forms” on page 20-24.

In addition, you can set up master/detail relationships in client datasets using nested tables. You can do this in one of two ways:

- Obtain records that contain nested details from a provider component. When a provider component represents the master table of a master/detail relationship, it automatically creates a nested dataset field to represent the details for each record.
- Define nested details using the Fields Editor. At design time, right click the client dataset and choose Fields Editor. Add a new persistent field to your client dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of your detail table.

When your client dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. Alternately, you can display and edit these datasets in data-aware controls by using a separate client dataset for the detail set. At design time, create persistent fields for the fields in your client dataset, including a DataSet field for the nested detail set.

You can now create a client dataset to represent the nested detail set. Set this detail client dataset’s *DataSetField* property to the persistent DataSet field in the master dataset.

In multi-tiered applications, using nested detail sets is necessary if you want to apply updates from master and detail tables to the application server. In flat-file database applications, using nested detail sets lets you save the details with the master records in one flat-file, rather than requiring you load two datasets separately, and then recreate the indexes to re-establish the master/detail relationship.

Note To use nested detail sets, the *ObjectView* property of the client dataset must be *True*.

Constraining data values

Client datasets provide support for data constraints. You can always supply custom constraints. This lets you provide your own, application-defined limits on what

values users post to a client dataset. For more information about supplying custom constraints, see “Adding custom constraints” on page 24-19.

In addition, if you are using a provider component to communicate with a remote database server, the provider has the option of supplying server constraints to the client dataset. For more information about controlling whether the application server communicates data constraints to a client dataset, see “Handling server constraints” on page 15-10.

In multi-tiered applications, there may be times when you want to turn off enforcement of data constraints, especially when the client dataset does not contain all of the records from the corresponding dataset on the application server. See “Handling constraints” on page 24-18 for more information on how and why to do this.

Making data read-only

TDataSet introduces the *CanModify* property so that applications can determine whether the data in a dataset can be edited. Applications can't change the *CanModify* property, because for some *TDataSet* descendants, the underlying database, not the application, controls whether data can be modified.

However, because client datasets represent in-memory data, your application can always control whether users can edit that data. To prevent users from modifying data, set the *ReadOnly* property of the client dataset to *True*. Setting *ReadOnly* to *True* sets the *CanModify* property to *False*.

Unlike other kinds of datasets, you do not need to close a client dataset to change its read-only status. An application can make a client dataset read-only or not on a temporary basis at any time merely by changing the current setting of the *ReadOnly* property.

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:

- When working with a provider, the change log is required by the mechanism for applying updates to the application server.
- In any application, the change log provides sophisticated support for undoing changes.

The *LogChanges* property enables you to disable logging temporarily. 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 single-tiered applications when you do not need 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 or not 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 single record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure to remove a change. If the removal occurs, *UndoLastChange* returns *False*. Use the *ChangeCount* property to determine whether there are any 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.
- 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 they are used in a stand-alone application or represent

data from a remote application server. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

Stand-alone 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” on page 24-25 describes this process.

You can’t use *MergeChangeLog* in multi-tiered applications. The application server needs the information in the change log so that it can resolve updated records with the data stored in the database. Instead, you call *ApplyUpdates*, which sends the changes to the application server and updates the *Data* property only when the modifications have been successfully posted to the database. See “Applying updates” on page 24-20 for more information about this process.

Sorting and indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They enable your application to set up relationships between client datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset is used in a multi-tiered application, it inherits a default index and sort order based on the data it receives from the application server. 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 for a dataset, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

Adding a new index

To create a new index for a client dataset, 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 specify options to make the entire index case sensitive or to sort in descending order. Alternately, you can provide a list of fields that should be sorted case-insensitively and a list of fields that should be sorted in descending order.
- The default level of grouping support for the index.

Tip You can index and sort on internally calculated fields with client datasets.

Indexes you create are sorted in ascending alphabetical order according to your machine's locale. In the index options parameter you can add *ixDescending* to the set of options to override this default.

Note When you create indexes at the same time as the client dataset, you can create indexes that sort in ascending order on some fields and descending order on others. See "Creating a dataset using field and index definitions" on page 13-15 for details.

Sorting of string fields in indexes is case sensitive by default. In the index options parameter you can add *ixCaseInsensitive* to ignore case when sorting.

Note When you create indexes at the same time as the client dataset, you can create indexes that are case insensitive on some fields and case sensitive on others. See "Creating a dataset using field and index definitions" on page 13-15 for details.

Warning Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

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 with a client dataset 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

SalesRep	Customer	OrderNo	Amount
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. When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index).

Client datasets allow you to 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 only display a field value 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
2	1	6	75
		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).

Indexing on the fly

Instead of creating an index that becomes part of the client dataset, you can create a temporary index on the fly by specifying fields to use for indexing in the *IndexFieldNames* property. Separate field names with semicolons. Ordering of field names in the list is significant.

Note When indexing on the fly, you can't specify a descending or case-insensitive index.

Warning Indexes created on the fly do not support grouping.

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. For more information about using calculated fields, see “Defining a calculated field” on page 19-8.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see “Using internally calculated fields in client datasets” below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see “Using maintained aggregates” on page 24-9.

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 *OnCalcFields* 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. If you are creating the client dataset using persistent fields, define fields as internally calculated by selecting *InternalCalc* in the Fields editor. If you are creating the client dataset using 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.

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 details on creating aggregated persistent fields, see "Defining an aggregate field" on page 19-11.

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 Table 24.1

Table 24.1 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 a client dataset. However, you can specify that you want to summarize over the records in a group instead. This allows you to 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. See “Using indexes to group data” on page 24-7 for information on grouping support.

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:

```

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.

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 an application server 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.

Copying data from another dataset

To copy the data from another dataset at design time, right click the dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one you want to copy from 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 an *OleVariant* in the form of a data packet. 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 uses a provider component, data packets are automatically assigned to *Data*. See "Using a client dataset with a data provider" on page 24-14 for more information on using providers with client datasets.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
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:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

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:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Cloning a client dataset cursor

TClientDataSet provides the *CloneCursor* procedure to enable you to work with a second view of a specified client dataset 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 interface.

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.

Using a client dataset with a data provider

When using a client dataset in a multi-tiered application, the client dataset obtains data from a provider on the application server and, after editing that data locally, applies updates to the remote database. It is also possible to use a client dataset with a provider that resides in the same application.

The following steps describe how to use a client dataset with a provider:

- 1 Specifying a data provider.
- 2 Optionally, Getting parameters from the application server or Passing parameters to the application server.
- 3 Optionally, Overriding the dataset on the application server.
- 4 Requesting data from an application server.
- 5 Handling constraints received from the application server.
- 6 Updating records.
- 7 Refreshing records.

In addition, client datasets allow you to communicate with the provider using a custom event.

Specifying a data provider

Before a client dataset can receive data from and apply updates to an application server, it must be associated with a dataset provider. To associate a client dataset with a provider in a multi-tiered application, use the *RemoteServer* and *ProviderName* properties. In single-tiered applications and in client applications used in briefcase mode as temporary single-tiered applications, these properties are not used. When using a client dataset with a provider that is instantiated in the same application, you do not need to use the *RemoteServer* property, but you can still use the *ProviderName* property, as long as the provider component has the same *Owner* as the client dataset.

RemoteServer specifies the name of a connection component from which to get a list of providers. The connection component resides in the same data module as the client dataset. It establishes and maintains a connection to an application server, sometimes called a “data broker”. For more information, see “The structure of the client application” on page 14-4.

At design time, after you specify *RemoteServer*, you can select a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This list also includes any local providers that belong to the same data module. At runtime, you can switch among available providers by setting *ProviderName* in code.

To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset’s *SetProvider* method.

Getting parameters from the application server

There are two circumstances when the client application needs to obtain parameter values from the application server:

- The client needs to know the value of output parameters on a stored procedure.
- The client wants to initialize the input parameters of a query or stored procedure to the current values of a query or stored procedure on the application server.

A client dataset stores parameter values in its *Params* property. These values are refreshed with any output parameters whenever the client dataset fetches data from the application server. However, there are times 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 application server by calling the *FetchParams* method. The parameters are returned in a data packet from the application server 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 The *FetchParams* method (or the Fetch Params command) will only work if the client dataset is connected to a provider whose associated dataset can supply parameters. *TDataSetProvider* can supply parameter values if it represents a query or stored procedure.

When working with 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 application server

Client datasets can pass parameters to the application server to specify what data they want provided in the data packets it sends. These parameters can specify

- 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 application server at design time or at runtime. At design time, select the client dataset, and then 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 sets the value of a parameter named *CustNo* to 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
  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*.

Sending query or stored procedure parameters

When the provider on the application server 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 application server 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 query or stored procedure. The application server then runs the 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 query or stored procedure component in the application server.

Limiting records with parameters

When the provider on the application server represents the results of a table component, you can use *Params* property to limit the records that are provided to the *Data* property.

Each parameter name must match the name of a field in the *TTable* component on the application server. The provider component on the application server sends only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider a client 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 server table is called) whose value identifies the customer whose orders it will display. When the client dataset requests data from the application server, it passes this parameter value. The application server then sends only the records for the identified customer. This is more efficient than letting the application server send all the orders records to the client application and then filtering the records on the client side.

Overriding the dataset on the application server

Usually, the provider on the application server is associated with a dataset that determines what data is supplied to clients. This dataset may have a property that specifies an SQL statement to generate the data, or it may represent a specific database table or stored procedure.

If the provider allows, the client dataset can override the property that indicates what data the dataset represents. To do so, you can set the client dataset's *CommandText* property. *CommandText* contains an SQL statement that replaces the SQL on the provider's dataset, or it contains the name of a table or stored procedure that replaces the table or stored procedure that the dataset currently represents. This allows the client dataset to specify dynamically what data it wants to see.

By default, provider's do not allow client datasets to specify a *CommandText* value in this way. To allow the client dataset to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider on the application server. Otherwise, the value of *CommandText* is ignored.

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 application server, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to the application server.

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.

Requesting data from an application server

The following table lists the properties and methods of *TClientDataSet* that determine how data is fetched from an application server in a multi-tiered application:

Table 24.2 Client datasets properties and method for handling data requests

Property or method	Purpose
FetchOnDemand property	Determines whether or not a client dataset automatically fetches data as needed, or relies on the application to call the client dataset's <i>GetNextPacket</i> , <i>FetchBlobs</i> , and <i>FetchDetails</i> functions to retrieve additional data.
PacketRecords property	Specifies the type or number of records to return in each data packet.
GetNextPacket method	Fetches the next data packet from the application server.
FetchBlobs method	Fetches any BLOB fields for the current record when the application server does not include BLOB data automatically.
FetchDetails method	Fetches nested detail datasets for the current record when the application server does not include these in data packets automatically.

By default, a client dataset retrieves all records from the application server. You can control how data is retrieved using *PacketRecords* and *FetchOnDemand*.

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 application is first opened, or the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after it first fetches data, a client dataset 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:

```
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. By default, the client dataset calls *GetNextPacket* to fetch data as needed. 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 only works if the remote data module preserves state information. That is, you must not be using MTS, and the remote data module must be configured so that each client application has its own data module instance. See “Supporting state information in remote data modules” on page 14-26 for information on how to use incremental fetching in stateless remote data modules.

You can also use *PacketRecords* to fetch metadata information about a database from the application server. To retrieve metadata information, set *PacketRecords* to *0*.

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), automatic fetching is enabled. To prevent automatic fetching of records as needed, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *false*, the application must explicitly call *GetNextPacket* to fetch records.

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 application server.

Handling constraints

Client datasets support two types of constraints. These are

- constraints that are sent from the application server in data packets.
- custom constraints provided by the client application.

Handling constraints from the server

By default, server constraints and default expressions are passed to client datasets by the application server, where they can be imposed on user data editing. When constraints are in effect, user edits to data in a client application that would violate server constraints are enforced on the client side, and are never passed to the application server for eventual rejection by the database server. This means that fewer updates generate error conditions during the updating process.

While importing of server constraints and expressions is an extremely valuable feature that enables a developer to preserve data integrity across platforms and applications, there may be times when an application needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value in a field, but the client dataset fetches multiple packets of records, the current maximum value in a field on the client may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client application 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 a client dataset's *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.

Note *DisableConstraints* and *EnableConstraints* control whether the client dataset applies constraints to its data. However, they have no effect on whether the application server includes constraint information in data packets. You can prevent the server from sending constraints in the first place using the provider's *Constraints* property. For more information on handling constraints from the server side, see "Handling server constraints" on page 15-10. For more information on working with the constraints once they have been imported, see "Using server constraints" on page 19-22.

Adding custom constraints

You can use the properties of the client dataset's field components to constrain the available values beyond those constraints that are supplied in data packets from the server. 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 application server also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is returned to the application server.
- 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” on page 19-22.

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 to add your own constraints that are checked when you post records.

Updating records

When a client application is connected to an application server, client datasets work with a local copy of data passed to them by the application server. The user sees and edits these copies in the client application’s data-aware controls. If server constraints are enabled on the client dataset, a user’s edits are constrained accordingly. User changes are temporarily stored by the client dataset in an internally maintained change log. The contents of the change log are stored as a data packet in the *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database.

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 application server. *Delta* is a data packet that contains a client dataset’s updated, inserted, and deleted records.
- 2 The application server’s provider component applies the updates to the database, caching any problem records that it can’t resolve at the server level. See “Responding to client update requests” on page 15-5 for details on how the server applies updates.
- 3 The application server’s provider component returns all unresolved records to the client 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 application 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 application server until the client application calls the *ApplyUpdates* method for the dataset. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the application server.

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the application server should tolerate before aborting the update process. If *MaxErrors* is 0, then as soon as an update error occurs on the application server, 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 value is set to indicate the number of records it could not write to the database. The application server also returns those records to the client dataset in the dataset.

The client dataset is responsible for reconciling records that generate errors. *ApplyUpdates* calls the *Reconcile* method to write updates to the database. *Reconcile* is an error-handling routine that indirectly calls the *ApplyUpdates* function of a provider component on the application server. The provider component's *ApplyUpdates* function writes the updates to the database and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset's *Reconcile* method. *Reconcile* then attempts to correct any remaining errors by calling the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For more information about creating and using *OnReconcileError*, see "Reconciling update errors" on page 24-21.

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.

Note If you are using MTS transactions or sharing an application server instance with other clients, you may want to communicate with the provider on the application server about persistent state information before or after you apply updates. The client dataset 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), the client dataset receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

Reconciling update errors

The provider on the application server returns error records and error information to the client dataset in a result data packet. If the application server returns an error count greater than zero, then for each record in the result data packet, the client dataset's *OnReconcileError* event occurs.

You should always code the *OnReconcileError* event handler, even if only to discard the records returned by the application server. The *OnReconcileError* event handler is passed four parameters:

- *DataSet*: the client dataset to which updates are applied. You can use client dataset methods to obtain information about problem records and to make changes to 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 an *OnReconcileError* event handler.

- *E*: An *EReconcileError* 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 lets you indicate what action to take when the *OnReconcileError* handler exits. On entry into the handler, *Action* is set to the action taken by the resolution process on the server. In your event handler, you set this parameter to
 - Skip this record, leaving it in the change log. (*raSkip*)
 - Stop the entire reconcile operation. (*raAbort*)
 - Merge the modification that failed into the corresponding record from the server. (*raMerge*) This only works if the server did not change any of the fields modified by the user.
 - Replace the current update in the change log with the value of the record in the event handler (which has presumably been corrected). (*raCorrect*)
 - Back out the changes for this record on the client dataset, reverting to the originally provided values. (*raCancel*)
 - Update the current record value to match the record on the server. (*raRefresh*)

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the *RecError* unit which ships in the object repository directory. (To use this dialog, add *RecError* to your uses clause.)

```

procedure TForm1.ClientDataSetReconcileError(DataSet: TClientDataSet; E: EReconcileError;
UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
    Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

```

Refreshing records

Client applications work with an in-memory snapshot of the data on the application server. As time elapses, other users may modify that data, so that the data in the client application becomes a less and 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 applications can also update the data while leaving the change log intact. To do this, call the client dataset's *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 application server but leaves any changes in the change log.

Warning It may not always be appropriate to call *RefreshRecord*. If the user's edits conflict with changes to the underlying dataset made by other users, calling *RefreshRecord* will mask this conflict. When the client application applies its updates, no reconcile error will occur and the application can't resolve the conflict.

In order to avoid masking update errors, client applications may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following code raises an exception if an attempt is made to refresh a modified record:

```
if ClientDataSet1.UpdateStatus <> usUnModified then
  raise Exception.Create('You must apply updates before refreshing the current record.');
```

ClientDataSet1.RefreshRecord;

Communicating with providers using custom events

Client datasets provide many opportunities for customizing the communication between the client application and the application server. Before and after every *IAppServer* method call that is directed at the client dataset's provider, the client dataset receives special events that are designed to allow the client dataset 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.
- 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 allow you to customize the communication between client and server.

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 *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Using a client dataset with flat-file data

Client datasets can function independently of a provider, such as in flat-file data base applications and “briefcase model” applications. When there is no provider, however, 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 get their data from a provider component:

- You can copy an existing dataset (at design or runtime). See “Copying data from another dataset” on page 24-12 for more information about copying existing datasets.
- You can define and create a new client dataset by creating persistent fields for the dataset and then choosing Create Dataset from its context menu. See “Creating a new dataset using persistent fields” on page 13-14 for details.
- You can define and create a new client dataset based on field definitions and index definitions. See “Creating a dataset using field and index definitions” on page 13-15 for details.

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. For more information about saving data to a file or stream, see “Saving data to a file or stream” on page 24-25.

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.

Merging changes into data

When you edit the data in a client dataset, the changes you make are recorded in the change log, but the changes do not affect the original version of the data.

To make your changes permanent, call *MergeChangeLog*. *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 applications that are connected to an application server. In this case, call *ApplyUpdates* to write changes to the database. For more information, see “Applying updates” on page 24-20.

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 and resolver. For an example of how to do this, see “Assigning data directly” on page 24-12.

Saving data to a file or stream

If you use a client dataset in a single-tiered application, then when you edit data and merge it, the changes you make exist only in memory. To make a permanent record of your changes, you must write them to disk. You can save the data 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.

If you always save the data to the same file, you can use the *FileName* property 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.

Note *SaveToFile* does not preserve any indexes you added to the client dataset.

Working with cached updates

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache 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.

This chapter describes when and how to use cached updates. It also describes the *TUpdateSQL* component that can be used in conjunction with cached updates to update virtually any dataset, particularly datasets that are not normally updatable.

Deciding when to use cached updates

Cached updates are primarily intended to reduce data access contention on remote database servers by

- Minimizing transaction times.
- Minimizing network traffic.

While cached updates can minimize transaction times and drastically reduce network traffic, they may not be appropriate for all database client applications that work with remote servers. There are three areas of consideration when deciding to use cached updates:

- **Cached data is local to your application, and is not under transaction control.** In a busy client/server environment this has two implications for your application:
 - Other applications can access and change the actual data on the server while your users edit their local copies of the data.
 - Other applications cannot see any data changes made by your application until it applies all its changes.

- **In master/detail relationships managing the order of applying cached updates can be tricky.** This is particularly true when there are nested master/detail relationships where one detail table is the master table for yet another detail table and so on.
- **Applying cached updates to read-only query-based datasets requires use of update objects.**

The data access components provide cached update methods and transaction control methods you can use in your application code to handle these situations, but you must take care that you cover all possible scenarios your application is likely to encounter in your working environment.

Using cached updates

This section provides a basic overview of how cached updates work in an application. If you have not used cached updates before, this process description serves as a guideline for implementing cached updates in your applications.

To use cached updates, the following order of processes must occur in an application:

- 1 Enable cached updates.** Enabling cached updates causes a read-only transaction that fetches as much data from the server as is necessary for display purposes and then terminates. Local copies of the data are stored in memory for display and editing. For more information about enabling and disabling cached updates, see “Enabling and disabling cached updates” on page 25-3.
- 2 Display and edit the local copies of records,** 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. For more information about displaying and editing when cached updates are enabled, see “Applying cached updates” on page 25-4.
- 3 Fetch additional records as necessary.** As a user scrolls through records, additional records are fetched as needed. Each fetch occurs within the context of another short duration, read-only transaction. (An application can optionally fetch all records at once instead of fetching many small batches of records.) For more information about fetching all records, see “Fetching records” on page 25-3.
- 4 Continue to display and edit local copies of records** until all desired changes are complete.
- 5 Apply the locally cached records to the database** or cancel the updates. For each record written to the database, an *OnUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event is triggered which 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 “Applying cached updates” on page 25-4.

If instead of applying updates, an application cancels updates, the locally cached copy of the records and all changes to them are freed without writing the changes

to the database. For more information about canceling updates, see “Canceling pending cached updates” on page 25-7.

Enabling and disabling cached updates

Cached updates are enabled and disabled through the *CachedUpdates* properties of *TTable*, *TQuery*, and *TStoredProc*. *CachedUpdates* is *False* by default, meaning that cached updates are not enabled for a dataset.

Note Client datasets always cache updates. They have no *CachedUpdates* property because you cannot disable cached updates on a client dataset.

To use cached updates, set *CachedUpdates* to *True*, either at design time (through the Object Inspector), or at runtime. When you set *CachedUpdates* to *True*, the dataset’s *OnUpdateRecord* event is triggered if you provide it. For more information about the *OnUpdateRecord* event, see “Creating an *OnUpdateRecord* event handler” on page 25-22.

For example, the following code enables cached updates for a dataset at runtime:

```
CustomersTable.CachedUpdates := True;
```

When you enable cached updates, a copy of all records necessary for display and editing purposes 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 current cache of local changes is applied to the database. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

Note 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.

To disable cached updates for a dataset, set *CachedUpdates* to *False*. If you disable cached updates when there are pending changes that you have not yet applied, those changes are discarded without notification. Your application can test the *UpdatesPending* property for this condition before disabling cached updates. For example, the following code prompts for confirmation before disabling cached updates for a dataset:

```
if (CustomersTable.UpdatesPending)
  if (Application.MessageBox("Discard pending updates?",
    "Unposted changes",
    MB_YES + MB_NO) = IDYES) then
    CustomersTable.CachedUpdates = False;
```

Fetching records

By default, when you enable cached updates, BDE datasets automatically handle fetching of data from the database when necessary. Datasets fetch enough records for display. During the course of processing, many such record fetches may occur. If your application has specific needs, it can fetch all records at one time. You can fetch all records by calling the dataset’s *FetchAll* method. *FetchAll* creates an in-memory,

local copy of all records from the dataset. If a dataset contains many records or records with large BLOB fields, you may not want to use *FetchAll*.

Client datasets use the *PacketRecords* property to indicate the number of records that should be fetched at any time. If you set the *FetchOnDemand* property to *True*, the client dataset automatically handles fetching of data when necessary. Otherwise, you can use the *GetNextPacket* method to fetch records from the data server. For more information about fetching records using a client dataset, see “Requesting data from an application server” on page 24-17.

Applying cached updates

When a dataset is in cached update mode, changes to data are not actually written to the database until your application explicitly calls methods that apply those changes. Normally an application applies updates in response to user input, such as through a button or menu item.

Important To apply updates to a set of records retrieved by an SQL query that does not return a live result set, you must use a *TUpdateSQL* object to specify how to perform the 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. For more information, see “Updating a read-only result set” on page 25-21. For more information about creating and using an *OnUpdateRecord* event handler, see “Creating an *OnUpdateRecord* event handler” on page 25-22.

Applying updates is a two-phase process that should occur in the context of a database component’s transaction control to enable your application to recover gracefully from errors. For more information about transaction handling with database components, see “Understanding database and session component interactions” on page 17-9.

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.

If the database write is unsuccessful:

- Database changes are rolled back, ending the database transaction.
- Cached updates are not committed, leaving them intact in the internal cache buffer.

If the database write is successful:

- Database changes are committed, ending the database transaction.
- Cached updates are committed, clearing the internal cache buffer (phase 2).

The two-phased approach to applying cached updates allows for effective error recovery, especially when updating multiple datasets (for example, the datasets associated with a master/detail form). For more information about handling update errors that occur when applying cached updates, see “Handling cached update errors” on page 25-23.

There are actually two ways to apply updates. To apply updates for a specified set of datasets associated with a database component, call the database component’s *ApplyUpdates* method. To apply updates for a single dataset, call the dataset’s *ApplyUpdates* and *Commit* methods. These choices, and their strengths, are described in the following sections.

Applying cached updates with a database component method

Ordinarily, applications cache updates at the dataset level. However, there are times when it is important to apply the updates to multiple interrelated datasets in the context of a single transaction. For example, when working with master/detail forms, you will likely want to commit changes to master and detail tables together.

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:

```
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;
```

The above sequence starts a transaction, and writes cached updates to the database. If successful, it also commits the transaction, and then commits the cached updates. If unsuccessful, this method rolls back the transaction, and does not change the status of the cached updates. In this latter case, your application should handle cached update errors through a dataset’s *OnUpdateError* event. For more information about handling update errors, see “Handling cached update errors” on page 25-23.

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 used in a master/detail form:

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

For more information about updating master/detail tables, see “Applying updates for master/detail tables” on page 25-6.

Applying cached updates with dataset component methods

You can apply updates for individual 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).

Applying updates at the dataset level gives you control over the order in which updates are applied to individual datasets. Order of update application is especially critical for handling master/detail relationships. To ensure the correct ordering of updates for master/detail tables, you should always apply updates at the dataset level. For more information see "Applying updates for master/detail tables" on page 25-6.

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset previously used to illustrate updates through a database method:

```

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;

```

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 re-raises 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.

Applying updates for master/detail tables

When you apply updates for master/detail tables, the order in which you list datasets to update is significant. Generally you should always update master tables before detail tables, except when handling deleted records. In complex master/detail relationships where the detail table for one relationship is the master table for another detail table, the same rule applies.

You can update master/detail tables at the database or dataset component levels. For purposes of control (and of creating explicitly self-documented code), you should

apply updates at the dataset level. The following example illustrates how you should code cached updates to two tables, *Master* and *Detail*, involved in a master/detail relationship:

```
Database1.StartTransaction;
try
    Master.ApplyUpdates;
    Detail.ApplyUpdates;
    Database1.Commit;
except
    Database1.Rollback;
    raise;
end;
Master.CommitUpdates;
Detail.CommitUpdates;
```

If an error occurs during the application of updates, this code also leaves both the cache and the underlying data in the database tables in the same state they were in before the calls to *ApplyUpdates*.

If an exception is *raised* during the call to *Master.ApplyUpdates*, it is handled like the single dataset case previously described. Suppose, however, that the call to *Master.ApplyUpdates* succeeds, and the subsequent call to *Detail.ApplyUpdates* fails. In this case, the changes are already applied to the master table. Because all data is updated inside a database transaction, however, even the changes to the master table are rolled back when *Database1.Rollback* is called in the *except* block. Furthermore, *UpdatesMaster.CommitUpdates* is not called because the exception which is re-raised causes that code to be skipped, so the cache is also left in the state it was before the attempt to update.

To appreciate the value of the two-phase update process, assume for a moment that *ApplyUpdates* is a single-phase process which updates the data *and the cache*. If this were the case, and if there were an error while applying the updates to the *Detail* table, then there would be no way to restore both the data and the cache to their original states. Even though the call to *Database1.Rollback* would restore the database, there would be no way to restore the cache.

Canceling pending cached updates

Pending cached updates are updated records that are posted to the cache but not yet applied to the database. There are three ways to cancel pending cached updates:

- To cancel all pending updates and disable further cached updates, set the *CachedUpdates* property to *False*.
- To discard all pending updates without disabling further cached updates, call the *CancelUpdates* method.
- To cancel updates made to the current record call *RevertRecord*.

The following sections discuss these options in more detail.

Canceling pending updates and disabling further cached updates

To cancel further caching of updates and delete all pending cached updates without applying them, set the *CachedUpdates* property to *False*. When *CachedUpdates* is set to *False*, the *CancelUpdates* method is automatically invoked.

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted record simply disappear.

Note This option is not available for client datasets.

Canceling pending cached updates

CancelUpdates clears the cache of all pending updates, and restores the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. For example, the following statement cancels updates for the *CustomersTable*:

```
CustomersTable.CancelUpdates;
```

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted records simply disappear.

Note Calling *CancelUpdates* does not disable cached updating. It only cancels currently pending updates. To disable further cached updates, set the *CachedUpdates* property to *False*.

Canceling updates to the current record

RevertRecord restores the current record in the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. It is most frequently used in an *OnUpdateError* event handler to correct error situations. For example,

```
CustomersTable.RevertRecord;
```

Undoing cached changes to one record does not affect any other records. If only one record is in the cache of updates and the change is undone using *RevertRecord*, the *UpdatesPending* property for the dataset component is automatically changed from *True* to *False*.

If the record is not modified, this call has no effect. For more information about creating an *OnUpdateError* handler, see “Creating an OnUpdateRecord event handler” on page 25-22.

Undeleting cached records

To undelete a cached record requires some coding because once the deleted record is posted to the cache, it is no longer the current record and no longer even appears in the dataset. In some instances, however, you may want to undelete such records. The process involves using the *UpdateRecordTypes* property to make the deleted records “visible,” and then calling *RevertRecord*. Here is a code example that undeletes all deleted records in a table:

```

procedure TForm1.UndeleteAll(DataSet: TBDEDataSet)
begin
  DataSet.UpdateRecordTypes := [rtDeleted];           { show only deleted records }
  try
    DataSet.First;                                   { go to the first previously deleted record }
    while not (DataSet.Eof)
      DataSet.RevertRecord;                           { undelete until we reach the last record }
    except
      { restore updates types to recognize only modified, inserted, and unchanged }
      DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
      raise;
    end;
    DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
  end;

```

Specifying visible records in the cache

The *UpdateRecordTypes* property controls what type of records are visible in the cache when cached updates are enabled. *UpdateRecordTypes* works on cached records in much the same way as filters work on tables. *UpdateRecordTypes* is a set, so it can contain any combination of the following values:

Table 25.1 TUpdateRecordType values

Value	Meaning
<i>rtModified</i>	Modified records
<i>rtInserted</i>	Inserted records
<i>rtDeleted</i>	Deleted records
<i>rtUnmodified</i>	Unmodified records

The default value for *UpdateRecordTypes* includes only *rtModified*, *rtInserted*, and *rtUnmodified*, with deleted records (*rtDeleted*) not displayed.

The *UpdateRecordTypes* property is primarily useful in an *OnUpdateError* event handler for accessing deleted records so they can be undeleted through a call to *RevertRecord*. This property is also useful if you wanted to provide a way in your application for users to view only a subset of cached records, for example, all newly inserted (*rtInserted*) records.

For example, you could have a set of four radio buttons (*RadioButton1* through *RadioButton4*) with the captions All, Modified, Inserted, and Deleted. With all four radio buttons assigned to the same *OnClick* event handler, you could conditionally display all records (except deleted, the default), only modified records, only newly inserted records, or only deleted records by appropriately setting the *UpdateRecordTypes* property.

```

procedure TForm1.UpdateFilterRadioButtonsClick(Sender: TObject);
begin
  if RadioButton1.Checked then
    CustomerQuery.UpdateRecordTypes := [rtUnmodified, rtModified, rtInserted]
  else if RadioButton2.Checked then

```

```

    CustomerQuery.UpdateRecordTypes := [rtModified]
  else if RadioButton3.Checked then
    CustomerQuery.UpdateRecordTypes := [rtInserted]
  else
    CustomerQuery.UpdateRecordTypes := [rtDeleted];
end;

```

For more information about creating an *OnUpdateError* handler, see “Creating an *OnUpdateRecord* event handler” on page 25-22.

Checking update status

When cached updates are enabled for your application, you can keep track of each pending update record in the cache by examining the *UpdateStatus* property for the record. Checking update status is most frequently used in *OnUpdateRecord* and *OnUpdateError* event handlers. For more information about creating and using an *OnUpdateRecord* event, see “Creating an *OnUpdateRecord* event handler” on page 25-22. For more information about creating and using an *OnUpdateError* event, see “Handling cached update errors” on page 25-23.

As you iterate through a set of pending changes, *UpdateStatus* changes to reflect the update status of the current record. *UpdateStatus* returns one of the following values for the current record:

Table 25.2 Return values for UpdateStatus

Value	Meaning
<i>usUnmodified</i>	Record is unchanged
<i>usModified</i>	Record is changed
<i>usInserted</i>	Record is a new record
<i>usDeleted</i>	Record is deleted

When a dataset is first opened all records will have an update status of *usUnmodified*. As records are inserted, deleted, and so on, the status values change. Here is an example of *UpdateStatus* property used in a handler for a dataset’s *OnScroll* event. The event handler displays the update status of each record in a status bar.

```

procedure TForm1.CustomerQueryAfterScroll(DataSet: TDataSet);
begin
  with CustomerQuery do begin
    case UpdateStatus of
      usUnmodified: StatusBar1.Panels[0].Text := 'Unmodified';
      usModified: StatusBar1.Panels[0].Text := 'Modified';
      usInserted: StatusBar1.Panels[0].Text := 'Inserted';
      usDeleted: StatusBar1.Panels[0].Text := 'Deleted';
      else StatusBar1.Panels[0].Text := 'Undetermined status';
    end;
  end;
end;

```

Note If a record’s *UpdateStatus* is *usModified*, you can examine the *OldValue* property for each field in the dataset to determine its previous value. *OldValue* is meaningless for

records with *UpdateStatus* values other than *usModified*. For more information about examining and using *OldValue*, see “Accessing a field’s *OldValue*, *NewValue*, and *CurValue* properties” on page 25-26.

Using update objects to update a dataset

TUpdateSQL is an update component that uses SQL statements to update a dataset. You must provide one *TUpdateSQL* component for each underlying table accessed by the original query that you want to update.

Note If you use more than one update component to perform an update operation, you must create an *OnUpdateRecord* event to execute each update component.

An update component 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 the *UpdateKind* parameter automatically generated on a record update event. *UpdateKind* specifies 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.

Specifying the UpdateObject property for a dataset

One or more update objects can be associated with a dataset to be updated. Associate update objects with the update dataset either by setting the dataset component’s *UpdateObject* property to the update object or by setting the update object’s *DataSet* property to the update dataset. Which method is used depends on whether only one base table in the update dataset is to be updated or multiple tables.

You must use one of these two means of associating update datasets with update objects. Without proper association, the dynamic filling of parameters in the update object’s SQL statements cannot occur. Use one association method or the other, but never both.

How an update object is associated with a dataset also determines how the update object is executed. An update object might be executed automatically, without explicit intervention by the application, or it might need to be explicitly executed. If the association is made using the dataset component's *UpdateObject* property, the update object will automatically be executed. If the association is made with the update object's *DataSet* property, you must programmatically execute the update object.

The sections that follow explain the process of associating update objects with update dataset components in greater detail, along with suggestions about when each method should be used and effects on update execution.

Using a single update object

When only one of the base tables referenced in the update dataset needs to be updated, associate an update object with the dataset by setting the dataset component's *UpdateObject* property to the name of the update object.

```
Query1.UpdateObject := UpdateSQL1;
```

The update SQL statements in the update object are automatically executed when the update dataset's *ApplyUpdates* method is called. The update object is invoked for each record that requires updating. Do not call the update object's *ExecSQL* method in a handler for the *OnUpdateRecord* event as this will result in a second attempt to apply each record's update.

If you supply a handler for the dataset's *OnUpdateRecord* event, the minimum action that you need to take in that handler is setting the event handler's *UpdateAction* parameter to *uaApplied*. You may optionally perform data validation, data modification, or other operations like setting parameter values.

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 the dataset by setting its *DataSet* property to the name of the dataset. The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
```

The update SQL statements in the update object are not automatically executed when the update dataset's *ApplyUpdates* method is called. To update records, you must supply a handler for the dataset component's *OnUpdateRecord* event and call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating.

In the handler for the dataset's *OnUpdateRecord* event, the minimum actions that you need to take in that handler are:

- Calling the update object's *SetParams* method (if you later call *ExecSQL*).

- Executing the update object for the current record with *ExecSQL* or *Apply*.
- Setting the event handler's *UpdateAction* parameter to *uaApplied*.

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Note It is also possible to have one update object associated with the dataset using the dataset component's *UpdateObject* property, and the second and subsequent update objects associated using their *DataSet* properties. The first update object is executed automatically on calling the dataset component's *ApplyUpdates* method. The rest need to be manually executed.

Creating SQL statements for update components

To update a record in an associated dataset, an update object uses one of three SQL statements. The three SQL statements delete, insert, and modify records cached for update. The statements are contained in the update object's string list properties *DeleteSQL*, *InsertSQL*, and *ModifySQL*. As each update object is used to update a single table, the object's update statements each reference the same base table.

As the update for each record is applied, one of the three SQL statements is executed against the base table updated. Which SQL statement is executed depends on the *UpdateKind* parameter automatically generated for each record's update.

Creating the SQL statements for update objects can be done at design time or at runtime. The sections that follow describe the process of creating update SQL statements in greater detail.

Creating SQL statements at design time

To create the SQL statements for an update component,

- 1 Select the *TUpdateSQL* component.
- 2 Select the name of the update component from the drop-down list for the dataset component's *UpdateObject* property in the Object Inspector. 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.
- 3 Right-click the update component and select UpdateSQL Editor from the context menu to invoke the Update SQL editor. The editor creates SQL statements for the update component'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 may 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.

Important 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 than 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.

For more information about old and new value parameter substitution, see “Accessing a field’s OldValue, NewValue, and CurValue properties” on page 25-26.

Composing update SQL statements

The *TUpdateSQL* component has three properties for updating SQL statements: *DeleteSQL*, *InsertSQL*, and *ModifySQL*. As the names of the properties imply, these SQL statements 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 tables 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 a 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 an update component's Query property

Use the *Query* property of an update component to access one of the update SQL properties *DeleteSQL*, *InsertSQL*, or *ModifySQL*, such as to set or alter the SQL statement. Use *UpdateKind* constant values as an index into the array of query components. The *Query* property is only accessible at runtime.

The statement below uses the *UpdateKind* constant *ukDelete* with the *Query* property to access the *DeleteSQL* property.

```
with UpdatesQL1.Query[ukDelete] do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Normally, the properties indexed by the *Query* property are set at design time using the Update SQL editor. You might, however, need to access these values at runtime if you are generating a unique update SQL statement for each record and not using

parameter binding. The following example generates a unique *Query* property value for each row updated:

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with UpdateSQL1 do begin
    case UpdateKind of
      ukModified:
        begin
          Query[UpdateKind].Text := Format('update emptab set Salary = %d where EmpNo = %d',
            [EmpAuditSalary.NewValue, EmpAuditEmpNo.OldValue]);
          ExecSQL(UpdateKind);
        end;
      ukInserted:
        {...}
      ukDeleted:
        {...}
    end;
  end;
  UpdateAction := uaApplied;
end;

```

Note *Query* returns a value of type *TDataSetUpdateObject*. To treat this return value as a *TUpdateSQL* component, to use properties and methods specific to *TUpdateSQL*, typecast the *UpdateObject* property. For example:

```

with (DataSet.UpdateObject as TUpdateSQL).Query[UpdateKind] do begin
  { perform operations on the statement in DeleteSQL }
end;

```

For an example of using this property, see “Calling the SetParams method” on page 25-18.

Using the DeleteSQL, InsertSQL, and ModifySQL properties

Use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to set the update SQL statements for each. These properties are all string list containers. Use the methods of string lists to enter SQL statement lines as items in these properties. Use an integer value as an index to reference a specific line within the property. The *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties are accessible both at design time and at runtime.

```

with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;

```

Below, the third line of an SQL statement is altered using an index of 2 with the *ModifySQL* property.

```

UpdateSQL1.ModifySQL[2] := 'WHERE ItemNo = :ItemNo';

```

Executing update statements

There are a number of methods involved in executing the update SQL for an individual record update. These method calls are typically used within a handler for the *OnUpdateRecord* event of the update object to execute the update SQL to apply the current cached update record. The various methods are applicable under different circumstances. The sections that follow discuss each of the methods in detail.

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 Values for the record are bound to the parameters in the appropriate update SQL statement.
- 2 The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. Only use *Apply* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *Apply* automatically calls the *SetParams* method to bind old and new field values to specially named parameters in the update SQL statement. Do not call *SetParams* yourself when using *Apply*. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's *UpdateObject* property to associate dataset and update object, this method is called automatically. Do not call *Apply* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *Apply*.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  UpdateSQL1.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;

```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note The operations performed by *Apply* are analogous to the *SetParams* and *ExecSQL* methods described in the following sections.

Calling the SetParams method

The *SetParams* method for an update component uses special parameter substitution rules to substitute old and new field values into the update SQL statement.

Ordinarily, *SetParams* is called automatically by the update component's *Apply* method. If you call *Apply* directly in an *OnUpdateRecord* event, do not call *SetParams* yourself. If you execute an update object using its *ExecSQL* method, call *SetParams* to bind values to the update statement's parameters.

SetParams sets the parameters of the SQL statement indicated by the *UpdateKind* parameter. Only those parameters that use a special naming convention automatically have a value assigned. If the parameter has the same name as a field or the same name as a field prefixed with "OLD_" the parameter is automatically a value. Parameters named in other ways must be manually assigned values. For more information see the section "Understanding parameter substitution in update SQL statements" on page 25-14.

The following example illustrates one such use of *SetParams*:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with DataSet.UpdateObject as TUpdateSQL do begin
    SetParams(UpdateKind);
    if UpdateKind = ukModified then
      Query[UpdateKind].ParamByName('DateChanged').Value := Now;
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;
```

Note This example assumes that the *ModifySQL* property for the update component is as follows:

```
UPDATE EmpAudit
SET EmpNo = :EmpNo, Salary = :Salary, Changed = :DateChanged
WHERE EmpNo = :OLD_EmpNo
```

In this example, the call to *SetParams* supplies values to the *EmpNo* and *Salary* parameters. The *DateChanged* parameter is not set because the name does not match the name of a field in the dataset, so the next line of code sets this value explicitly.

Calling the ExecSQL method

The *ExecSQL* method for an update component manually applies updates for the current record. There are two steps involved in this process:

- 1 Values for the record are bound to the parameters in the appropriate update SQL statement.
- 2 The SQL statement is executed.

Call the *ExecSQL* method to apply the update for the current record in the update cache. Only use *ExecSQL* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *ExecSQL* does not automatically call the *SetParams* method to bind update SQL statement parameter values; call *SetParams* yourself before invoking *ExecSQL*. The *ExecSQL* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's *UpdateObject* property to associate dataset and update object, this method is called automatically. Do not call *ExecSQL* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *ExecSQL*.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with (DataSet.UpdateObject as TUpdateSQL) do begin
    SetParams(UpdateKind);
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;

```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note The operations performed by *ExecSQL* and *SetParams* are analogous to the *Apply* method described previously.

Using dataset components to update a dataset

Applying cached updates usually involves use of one or more update objects. The update SQL statements for these objects apply the data changes to the base table. Using update components is the easiest way to update a dataset, but it is not a requirement. You can alternately use dataset components like *TTable* and *TQuery* to apply the cached updates.

In a handler for the dataset component's *OnUpdateRecord* event, use the properties and methods of another dataset component to apply the cached updates for each record.

For example, the following code uses a table component to perform updates:

```

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
            Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;

```

```

ukInsert:
  begin
    Insert;
    UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
    Post;
  end;
ukModify: DeleteRecord;
end;
UpdateAction := uaApplied;
end;

```

Updating a read-only result set

Although the Borland Database Engine (BDE) attempts to provide an updatable, or “live” query result when the *RequestLive* property for a *TQuery* component is *True*, there are some situations where it cannot do so. (For more information see the section “Deciding when to use cached updates” on page 25-1.)

In these situations, you can manually update a dataset as follows:

- 1 Add a *TUpdateSQL* component to the data module in your application.
- 2 Set the dataset component’s *UpdateObject* property to the name of the *TUpdateSQL* component in the data module.
- 3 Enter the SQL update statement for the result set to the update component’s *ModifySQL*, *InsertSQL*, or *DeleteSQL* properties, or use the Update SQL editor.
- 4 Close the dataset.
- 5 Set the dataset component’s *CachedUpdates* property to *True*.
- 6 Reopen the dataset.

Note In many circumstances, you may also want to write an *OnUpdateRecord* event handler for the dataset.

Controlling the update process

When a dataset component’s *ApplyUpdates* method is called, an attempt is made to apply the updates for all records in the update cache 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, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

The sections that follow describe when you might need to provide a handler for the *OnUpdateRecord* event and how to create a handler for this event.

Determining if you need to control the updating process

Some of the time when you use cached updates, all you need to do is call *ApplyUpdates* to apply cached changes to the base tables in the database (for example, when you have exclusive access to a local Paradox or dBASE table through a *TTable* component). In most other cases, however, you either might want to or must provide additional processing to ensure that updates can be properly applied. Use a handler for the updated dataset component's *OnUpdateRecord* event to provide this additional processing.

For example, you might want to use the *OnUpdateRecord* event to provide validation routines that adjust data before it is applied to the table, or you might want to use the *OnUpdateRecord* event to provide additional processing for records in master and detail tables before writing them to the base tables.

In many cases you must provide additional processing. For example, if you access multiple tables using a joined query, then you must provide one *TUpdateSQL* object for each table in the query, and you must use the *OnUpdateRecord* event to make sure each update object is executed to write changes to the tables.

The following sections describe how to create and use an *TUpdateSQL* object and how to create and use an *OnUpdateRecord* event.

Creating an OnUpdateRecord event handler

The *OnUpdateRecord* event handles cases where a single update component cannot be used to perform the required updates, or when your application needs more control over special parameter substitution. The *OnUpdateRecord* event fires once for the attempt to apply the changes for each modified record in the update cache.

To create an *OnUpdateRecord* event handler for a dataset:

- 1 Select the dataset component.
- 2 Choose the Events page in the Object Inspector.
- 3 Double-click the *OnUpdateRecord* property value to invoke the code editor.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update to perform. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. When using an update component, you need to pass this parameter to its execution and parameter binding methods. For example using *ukModify* with the *Apply* method executes the update object's *ModifySQL* statement. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update to perform.

The *UpdateAction* parameter indicates if you applied an update or not. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. Unless you encounter a problem during updating, your event handler should set this parameter to *uaApplied* before exiting. If you decide not to update a particular record, set 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. For more information about *UpdateAction*, see “Specifying the action to take” on page 25-25.

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. For more information about *OldValue* and *NewValue* see “Accessing a field’s OldValue, NewValue, and CurValue properties” on page 25-26.

Important The *OnUpdateRecord* event, like the *OnUpdateError* and *OnCalcFields* event handlers, should never call any methods that change which record in a dataset is the current record.

Here is an *OnUpdateRecord* event handler that executes two update components using their *Apply* methods. The *UpdateKind* parameter is passed to the *Apply* method to determine which update SQL statement in each update object to execute.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  EmployeeUpdateSQL.Apply(UpdateKind);
  JobUpdateSQL.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

In this example the *DataSet* parameter is not used. This is because the update components are not associated with the dataset component using its *UpdateObject* property.

Handling cached update errors

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change the record in a database before your application applies its updates. Even if there is no conflict between user updates, errors can occur when a record’s update is applied. The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports an error.

A 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.

Caution Do not call any dataset methods that change the current record (such as *Next* and *Prior*) in an *OnUpdateError* event handler. Doing so causes the event handler to enter an endless loop.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

The following sections describe specific aspects of error handling using an *OnUpdateError* handler, and how the event's parameters are used.

Referencing the dataset to which to apply updates

DataSet references the dataset to which updates are applied. To process new and old record values during error handling you must supply this reference.

Indicating the type of update that generated an error

The *OnUpdateRecord* event receives the parameter *UpdateKind*, which of type *TUpdateKind*. It 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*:

Table 25.3 UpdateKind values

Value	Meaning
<i>ukModify</i>	Editing an existing record caused an error.
<i>ukInsert</i>	Inserting a new record caused an error.
<i>ukDelete</i>	Deleting an existing record caused an error.

The example below shows the decision construct to perform different operations based on the value of the *UpdateKind* parameter.

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  case UpdateKind of
    ukModify:
      begin
        { handle error due to applying record modification update }
      end;
    ukInsert:
      begin
        { handle error due to applying record insertion update }
      end;
    ukDelete:
      begin
        { handle error due to applying record deletion update }
      end;
  end;
end;
```

Specifying the action to take

UpdateAction is a parameter of type *TUpdateAction*. 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. *UpdateAction* can be set to one of the following values:

Table 25.4 UpdateAction values

Value	Meaning
<i>uaAbort</i>	Aborts the update operation without displaying an error message.
<i>uaFail</i>	Aborts the update operation, and displays an error message. This is the default value for <i>UpdateAction</i> when you enter an update error handler.
<i>uaSkip</i>	Skips updating the row, but leaves the update for the record in the cache.
<i>uaRetry</i>	Repeats the update operation. Correct the error condition before setting <i>UpdateAction</i> to this value.
<i>uaApplied</i>	Not used in error handling routines.

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).

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.

The *uaApplied* value should only be used inside an *OnUpdateRecord* event. Do not set this value in an update error handler. For more information about update record events, see “Creating an *OnUpdateRecord* event handler” on page 25-22.

Working with error message text

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:

```
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. For example, the following code checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ 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;
end;
```

Accessing a field's OldValue, NewValue, and CurValue properties

When cached updates are enabled for records, 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*. In client datasets, an additional *TField* property, *CurValue*, stores the field values that currently appear in the dataset. *CurValue* will be the same as *OldValue* unless another user has edited the record. If another user has edited the record, *CurValue* will give the current value of the field which was posted by that user.

These values provide the only way to inspect and change update values in *OnUpdateError* and *OnUpdateRecord* event handlers. For more information about *OnUpdateRecord*, see “Creating an OnUpdateRecord event handler” on page 25-22.

In some cases, you may be able to use the *OldValue*, *NewValue*, and *CurValue* properties to determine the cause of an error and correct it. For example, the following code handles corrections to a salary field which can only be increased by 25 percent at a time (enforced by a constraint on the server):

```
var
  SalaryDif: Integer;
  OldSalary: Integer;
begin
  OldSalary := EmpTabSalary.OldValue;
  SalaryDif := EmpTabSalary.NewValue - OldSalary;
  if SalaryDif / OldSalary > 0.25 then begin
    { Increase was too large, drop it back to 25% }
    EmpTabSalary.NewValue := OldSalary + OldSalary * 0.25;
    UpdateAction := uaRetry;
  end
  else
    UpdateAction := uaSkip;
end;
```

NewValue is decreased to a 25 percent increase in the case where the cached update would have increased salary by a larger percentage. Then the update operation is retried. To improve the efficiency of this routine, the *OldValue* parameter is stored in a local variable.

Using data controls

This chapter describes how to use data-aware visual controls to display and edit data associated with the tables and queries in your database application. A *data-aware* control derives display data from a database source outside the application, and can optionally post (or return) data changes to a data source.

This chapter describes basic features common to all data control components, including how and when to use individual components.

Most data-aware components represent information stored in a dataset. These can be grouped into controls that represent a single field and controls that represent sets of records, such as DBGrids and control grids. In addition, the navigator control, *TDBNavigator*, is a visual tool that lets your users navigate and manipulate records.

More complex data-aware controls for decision support are discussed in Chapter 27, “Using decision support components.”

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

You place data controls from the Data Controls page of the Component palette onto the forms in your database application. Data controls generally enable you to display and edit fields of data associated with the current record in a dataset. Table 26.1

summarizes the data controls that appear on the Data Controls page of the Component palette.

Table 26.1 Data controls

Data control	Description
<i>TDBGrid</i>	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.
<i>TDBNavigator</i>	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
<i>TDBText</i>	Displays data from a field as a label.
<i>TDBEdit</i>	Displays data from a field in an edit box.
<i>TDBMemo</i>	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
<i>TDBImage</i>	Displays graphics from a data field in a graphics box.
<i>TDBListBox</i>	Displays a list of items from which to update a field in the current data record.
<i>TDBComboBox</i>	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
<i>TDBCheckBox</i>	Displays a check box that indicates the value of a Boolean field.
<i>TDBRadioGroup</i>	Displays a set of mutually exclusive options for a field.
<i>TDBLookupListBox</i>	Displays a list of items looked up from another dataset based on the value of a field.
<i>TDBLookupComboBox</i>	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.
<i>TDBCtrlGrid</i>	Displays a configurable, repeating set of data-aware controls within a grid.
<i>TDBRichEdit</i>	Displays formatted data from a field in an edit box.

Data controls are data-aware at design time. When you set a control's *DataSource* property to an active data source while building an application, you can immediately see live data in the controls. 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" on page 19-5 in Chapter 19, "Working with field components."

At runtime, data controls display data and, if your application, the control, and the database 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 acts as a conduit between the control and a dataset containing data. It is described in more detail in "Using data sources" on page 26-5.

To associate a data control with a dataset,

- 1 Place a dataset and a data source in a data module (or on a form), and set their properties as appropriate.
- 2 Place a data control from the Data Access page of the Component palette onto a form.
- 3 Set the *DataSource* property of the control to the name of a data source component from which to get data.
- 4 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 within a dataset.
- 5 Set the *Active* property of the dataset to *True* to display data in the control.

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 permits it.

Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. The *AutoEdit* property of the data source to which a control is attached determines if the underlying dataset enters *dsEdit* mode when data in a control is modified in response to keyboard or mouse events. When *AutoEdit* is *True* (the default), *dsEdit* mode is set as soon as editing commences. If *AutoEdit* is *False*, you must provide a *TDBNavigator* control with an *Edit* button (or some other method) to permit users to set *dsEdit* state at runtime. For more information about *TDBNavigator*, see “Navigating and manipulating records” on page 26-29.

Editing data in a control

The *ReadOnly* property of a data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. To prevent users from editing data in a control, set *ReadOnly* to *True*.

Properties of the data source and dataset underlying a control also determine if the user can successfully edit data with a control and can post changes to the dataset.

The *Enabled* property of a data source determines if controls attached to a data source are able to display fields values from the dataset, and therefore also determine if a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

The *ReadOnly* property of the dataset determines if user edits can be posted to the dataset. If *False* (the default), changes are posted to the dataset. If *True*, the dataset is read-only.

Note An additional read-only, runtime property *CanModify* determines if a dataset can be modified. *CanModify* is set to *True* if a database permits write access. If *CanModify* is *False*, a dataset is read-only. Query components that perform inserts and updates are, by definition, able to write to an underlying database, provided that your application and user have sufficient write privileges to the database itself.

The following table summarizes the factors that determine if a user can edit data in a control and post changes to the database.

Table 26.2 Properties affecting editing in data controls

Component	Property				
Data control	ReadOnly	false	false	false	true
Data Source	Enabled	true	true	false	—
Dataset	ReadOnly	false	false	—	—
Dataset	CanModify	true	false	—	—
(database)	write access	Read/Write	Read-only	—	—
Can write to database?		Yes	No	No	No

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying field component in a 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 copied only 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 components 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.

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* in addition to the call to *EnableControls* outside

the **try...finally** block. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
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;
```

Refreshing data display

The *Refresh* method for a dataset flushes local buffers and refetches 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.

Important

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 accessing a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, a data source does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Using data sources

A *TDataSource* component is a nonvisual database component that acts as a conduit between a dataset and data-aware components on a form that enable the display, navigation, and editing of the data underlying the dataset. Each data-aware control needs to 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 You can also use data source components to link datasets in master-detail relationships.

Place a data source component in a data module or form just as you place other nonvisual database components. You should place at least one data source component for each dataset component in a data module or form.

Using TDataSource properties

TDataSource has only a few published properties. The following sections discuss these key properties and how to set them at design time and runtime.

Setting the DataSet property

The *DataSet* property specifies the dataset from which a data source component gets its data. At design time, you can select a dataset from the drop-down list in the Object Inspector. 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 *Customers* and *Orders*:

```
with CustSource do
begin
  if DataSet = 'Customers' then
    DataSet := 'Orders'
  else
    DataSet := 'Customers';
end;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on the two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

Setting the Name property

Name enables you to specify a meaningful name for a data source component that distinguishes it from all other data sources in your application. The name you supply for a data source component is displayed below the component's icon in a data module.

Generally, you should provide a name for a data source component that indicates the dataset with which it is associated. For example, suppose you have a dataset called *Customers*, and that you link a data source component to it by setting the data source component's *DataSet* property to "Customers." To make the connection between the dataset and data source obvious in a data module, you should set the *Name* property for the data source component to something like "CustomersSource."

Setting the Enabled property

The *Enabled* property determines if a data source component 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.

Setting the AutoEdit property

The *AutoEdit* property of *TDataSource* specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If *AutoEdit* is *True* (the default), the dataset automatically enters Edit state when a user types in a linked data-aware control. Otherwise, a dataset enters Edit state only when the application explicitly calls its *Edit* method. For more information about dataset states, see “Determining and setting dataset states” in Chapter 18, “Understanding datasets.”

Using TDataSource events

TDataSource has three event handlers associated with it:

- *OnDataChange*
- *OnUpdateData*
- *OnStateChange*

Using the OnDataChange event

An *OnDataChange* event occurs whenever the cursor moves to a new record. When an application calls *Next*, *Previous*, *Insert*, or any method that leads to a change in the cursor position, an *OnDataChange* is triggered.

This event is useful if an application is keeping components synchronized manually.

Using the OnUpdateData event

An *OnUpdateData* event occurs whenever the data in the current record is about to be updated. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the database.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

Using the OnStateChange event

An *OnStateChange* event occurs whenever the state for a data source's dataset changes. A dataset's *State* property records its current state. *OnStateChange* is useful for performing actions as a *TDataSource*'s state changes.

For example, during the course of a normal database session, a dataset's state changes frequently. To track these changes, you could write an *OnStateChange* event handler that displays the current dataset state in a label on a form. The following code illustrates one way you might code such a routine. At runtime, this code displays the current setting of the dataset's *State* property and updates it every time it changes:

```

procedure TForm1.DataSource1.StateChange(Sender:TObject);
var
    S:String;
begin
    case CustTable.State of
        dsInactive: S := 'Inactive';
        dsBrowse: S := 'Browse';
        dsEdit: S := 'Edit';    dsInsert: S := 'Insert';
        dsSetKey: S := 'SetKey';
    end;
    CustTableStateLabel.Caption := S;
end;

```

Similarly, *OnStateChange* can be used to enable or disable buttons or menu items based on the current state:

```

procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
    CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
    CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
    ...
end;

```

Controls that represent a single field

Many of the controls on the data controls page of the component palette represent a single field in a database table. Most of these controls are similar in appearance and function to standard Windows controls that you place on forms. 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.

Displaying data as labels

TDBText is a read-only control similar to the *TLabel* component on the Standard page of the Component palette and the *TStaticText* component on the Additional page. 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* and *TStaticText*.

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 set to *False*, and the control is too small, data display is truncated.

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 *TTable* 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 binary large object (BLOB) data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display memo fields from dBASE and Paradox tables and text data contained in 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. *TDBMemo* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well. You can use *TDBRichEdit* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields.

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 component that displays bitmapped graphics contained in BLOB data fields. It captures BLOB graphics images from a dataset, and stores them internally in the Windows.DIB format.

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. 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 data-aware versions of standard list box and combo box controls. These useful controls provide the user with a set of default data values to choose from at runtime.

Note Data-aware list and combo box can be linked only to data sources for table components. They do not work with query components.

The following table describes these controls.

Table 26.3 Data-aware list box and combo box controls

Data control	Description
TDBListBox	Displays a list of items from which a user can update a field in the current record. The list of display items is a property of the control.
TDBComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is a property of the control.
TDBLookupListBox	Displays a list of items from which a user can update a column in the current record. The list of display items is looked up in another dataset.
TDBLookupComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is looked up in another dataset.

Displaying and editing data in a list box

TDBListBox 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.

Use the String List editor at design time to create the list of items to display in the *Items* property. The *Height* property determines how many items are visible in the list box at one time. The *IntegralHeight* property controls the way the list box is displayed. If *IntegralHeight* is *False* (the default), the bottom of the list box is determined by the *ItemHeight* property, and the bottom item might not be completely displayed. If *IntegralHeight* is *True* the visible bottom item in the list box is fully displayed.

Displaying and editing data in a combo box

The *TDBComboBox* control 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.

The *Items* property of the component specifies the items contained in the drop-down list. At design time, use the String List Editor to populate the *Items* list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a control is linked to a field through its *DataField* property, it displays the value for the field in the current row, regardless of whether it appears in the *Items* list. 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

TDBLookupListBox and *TDBLookupComboBox* are data-aware controls that derive a list of display items from one of two sources:

- Lookup field defined for a dataset.
- Secondary data source, data field, and key.

In either case, a user is presented 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.

Specifying a list based on a lookup field

To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. For more information about defining a lookup field for a dataset, see "Defining a lookup field" on page 19-9 in Chapter 19, "Working with field components."

To specify a 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.

When you activate a table associated with a lookup list box control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

Specifying a list based on a secondary data source

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 *DataField* property. 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 list box control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

Setting lookup list and combo box properties

The following table lists important properties for lookup list and combo boxes.

Table 26.4 TDBLookupListBox and TDBLookupComboBox properties

Property	Purpose
DataField	Specifies the field in the master dataset which provides the key value to be looked up in the lookup dataset. This field is modified when a user selects a list box or combo box item. If <i>DataField</i> is set to a lookup field, the <i>KeyField</i> , <i>ListField</i> , and <i>ListSource</i> properties are not used.
DataSource	Specifies a data source for the control. If the selection in the control is changed, this dataset is placed in <i>dsEdit</i> mode.
KeyField	Specifies the field in the lookup dataset corresponding to <i>DataField</i> . The control searches for the <i>DataField</i> value in the <i>KeyField</i> of the lookup dataset. The lookup dataset should have an index on this field to facilitate lookups.
ListField	Specifies the field of the lookup dataset to display in the control.
ListSource	Specifies a data source for the secondary (lookup) dataset. The sort order of items displayed in the list box or combo box is determined by the index specified by the <i>IndexName</i> property of the lookup dataset. That index need not be the same one used by the <i>KeyField</i> property.
RowCount	TDBLookupListBox only. Specifies the number of lines of text to display in the list box. The height of the list box is adjusted to fit this row count exactly.
DropDownRows	TDBLookupComboBox only. Specifies the number of lines of text to display in the drop-down list.

Searching incrementally for list item values

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.

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:

```
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 consists of 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 number of 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 is 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.

Viewing and editing data with TDBGrid

A *TDBGrid* control enables you to view and edit records in a dataset in a tabular grid format.

Figure 26.1 TDBGrid control

The diagram shows a TDBGrid control with a table of data. The table has four columns: VendorName, Address1, City, and State. The first row is highlighted, and a record indicator points to it. Labels above the table indicate the current field and column titles.

VendorName	Address1	City	State
Cacor Corporation	161 Southfield Rd	Southfield	OH
Underwater	50 N 3rd Street	Indianapolis	IN
J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
Divers' Supply Shop	5208 University Dr	Macon	GA
Techniques	52 Dolphin Drive	Redwood City	CA
Perry Scuba	3443 James Ave	Hapeville	GA

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.
- 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 Chapter 19, “Working with field components.”
- The dataset’s *ObjectView* property setting for grids displaying ADT and array fields. See “Displaying ADT and array fields” on page 26-22.

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 *State* property of a grid’s *Columns* property indicates if 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 either

by persistent field components for the dataset displayed in the grid, or for datasets without persistent field components, by a default set of data display characteristics.

Using a grid control in its default state

If a grid's *Columns.State* property is *csDefault*, the appearance of records is determined primarily by the properties of the fields in the grid's dataset. 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.

Properties of dynamic columns are retained for as long as the associated field component exists. 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 control 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 enables you to 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 (such as the grid or associated field) 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.

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.

Once you assign a value to a column property, it no longer changes when its default source changes. For example, if you assign a string to the column title's caption, the title caption is 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. You can use a blank column to display bitmaps or bar charts that graphically depict some aspect of a record's data in an otherwise blank cell, for example. To do so, you must override the cells' default drawing method.

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 *SelectedIndex* property 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.

Determining the source of a column property at runtime

At runtime you can test a column's *AssignedValues* property to determine whether a column property gets its values from an associated field component, or is assigned its own value.

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:

```
DBGrid1.Columns.RestoreDefaults;
```

To add a persistent column call the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

You can delete a persistent column by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

Finally, assigning *csCustomized* to the *Column.State* property for a grid at runtime puts the grid into customized mode. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset.

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, set the *Caption* option for the *Title* property in the Object Inspector.
- 4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

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 Select the field to remove in the Columns list box.
- 2 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.

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 by dragging the column in the actual grid, just as you can at runtime.

Defining a lookup list column

To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field object in the dataset. For more information about creating lookup fields, see , “Defining a lookup field,” on page 19-9.

Once the lookup field is defined, set the column’s *FieldName* to the lookup field name and make sure the column’s *ButtonStyle* is set to *cbAuto*. The grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode. The drop-down list is populated with lookup values defined by the lookup field.

Defining a pick list column

A pick list column looks and operates like a lookup list column, except that the column’s field is a normal field and the drop-down list is populated with the list of values in the column’s *PickList* property instead of from a lookup table.

To define a pick list column:

- 1 Select the column in the *Columns* list box.

2 Set *ButtonStyle* to *cbsAuto*.

3 Double-click the *Picklist* property in the Object Inspector to bring up a string list editor.

In the String List editor, enter the list of values you want to appear in the drop-down list for this column. If the pick list contains data, the column becomes a pick list column.

Note To restore a column to its normal behavior, delete all the text from the Pick 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 which 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.

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.

Table 26.5 Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: <i>TField.Alignment</i> .
ButtonStyle	<i>cbsAuto</i> : (default) Displays a drop-down list if the associated field is a lookup field, or if the column's <i>PickList</i> property contains data. <i>cbsEllipsis</i> : Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's <i>OnEditButtonClick</i> event. <i>cbsNone</i> : 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. For text foreground color, see the font property. Default Source: <i>TDBGrid.Color</i> .
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.

Table 26.5 Column properties (continued)

Property	Purpose
Expanded	Specifies whether the column is expanded. Only applies to columns representing ADT or array fields.
FieldName	Specifies the field name that is associated with this column. This can be blank.
ReadOnly	<i>True</i> : The data in the column cannot be edited by the user. <i>False</i> : (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default Source: derived from <i>TField.DisplayWidth</i> .
Font	Specifies the font type, size, and color used to draw text in the column. Default Source: <i>TDBGrid.Font</i> .
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.

Table 26.6 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: <i>TField.DisplayLabel</i> .
Color	Specifies the background color used to draw the column title cell. Default Source: <i>TDBGrid.FixedColor</i> .
Font	Specifies the font type, size, and color used to draw text in the column title. Default Source: <i>TDBGrid.TitleFont</i> .

Restoring default values to a column

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.

Displaying ADT and array fields

Depending on the value of the dataset's *ObjectView* property, a grid displays ADT and array fields either flattened out, or in an object mode, where the field can be expanded and collapsed. When *ObjectView* is *True*, the object fields can be expanded and collapsed. When a field is expanded, each child field appears in its own column with a title bar, which are below the title bar of the ADT or array field itself. When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields. A column can be expanded and collapsed by clicking on the arrow in the title bar of the field, and by setting the Expanded

property of the column. When the dataset's *ObjectView* property is *False*, each child field appears in a separate column.

Table 26.7 Properties that affect the way ADT and array fields appear in a TDBGrid

Property	Object	Purpose
Expandable	TColumn	Specifies whether the column can be expanded to show child fields in separate, editable columns.
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 an 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.

Figure 26.2 shows the 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.

Figure 26.2 TDBGrid control with *ObjectView* set to *False*

ID_KEY	ADT child fields			Array child fields		
	NAME_ADT.FIRST	NAME_ADT.MIDDLE	NAME_ADT.LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARRAY[2]
1	Stephan		Wright	415-908-9875	902-786-1245	
2	Whitney	N	Long			
3	Chris	T	Scanlan	234-232-1343		

Figure 26.2 and 26.3 show the grid with an ADT field and an array field. Figure 26.3 shows the fields collapsed. In this state they cannot be edited. Figure 26.4 shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

Figure 26.3 TDBGrid control with *Expanded* set to *False*

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,,)

Figure 26.4 TDBGrid control with Expanded set to True

ID_KEY	ADT child field columns			Array child field columns			
	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			

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, double-click the *Options* property. The list of options that you can set appears in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, indicating that you can collapse the list of properties by double-clicking the *Options* property.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Table 26.8 Expanded TDBGrid Options properties

Option	Purpose
dgEditing	<i>True</i> : (Default). Enables editing, inserting, and deleting records in the grid. <i>False</i> : Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	<i>True</i> : When a field is selected, it is in Edit state. <i>False</i> : (Default). A field is not automatically in Edit state when selected.
dgTitles	<i>True</i> : (Default). Displays field names across the top of the grid. <i>False</i> : Field name display is turned off.
dgIndicator	<i>True</i> : (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. <i>False</i> : The indicator column is turned off.
dgColumnResize	<i>True</i> : (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying <i>TField</i> component. <i>False</i> : Columns cannot be resized in the grid.

Table 26.8 Expanded TDBGrid Options properties (continued)

Option	Purpose
dgColLines	<i>True</i> : (Default). Displays vertical dividing lines between columns. <i>False</i> : Does not display dividing lines between columns.
dgRowLines	<i>True</i> : (Default). Displays horizontal dividing lines between records. <i>False</i> : Does not display dividing lines between records.
dgTabs	<i>True</i> : (Default). Enables tabbing between fields in records. <i>False</i> : Tabbing exits the grid control.
dgRowSelect	<i>True</i> : The selection bar spans the entire width of the grid. <i>False</i> : (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	<i>True</i> : (Default). The selection bar in the grid is always visible, even if another control has focus. <i>False</i> : The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	<i>True</i> : (Default). Prompt for confirmation to delete records (<i>Ctrl+Del</i>). <i>False</i> : Delete records without confirmation.
dgCancelOnExit	<i>True</i> : (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. <i>False</i> : Permits pending inserts.
dgMultiSelect	<i>True</i> : Allows user to select noncontiguous rows in the grid using <i>Ctrl+Shift</i> or <i>Shift+ arrow</i> keys. <i>False</i> : (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, Delphi raises an exception, and does not modify the record.

You can cancel all edits for a record by pressing *Esc* in any field before moving to another record.

Rearranging column order at design time

In grid controls with persistent columns, and default grids whose datasets contain persistent fields, you can reorder the grid columns at design time by clicking on the title cell of a column and dragging it to its new location in the grid.

Note Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Important 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.

Rearranging column order at runtime

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 the field components in the dataset underlying the grid. The order of fields in the physical table is not affected.

A grid's *OnColumnMoved* event is fired after a column has been moved.

To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

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.

Table 26.9 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.
OnDbClick	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 <i>State</i> is <i>csDefault</i> .
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 *OnDbClick* 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.

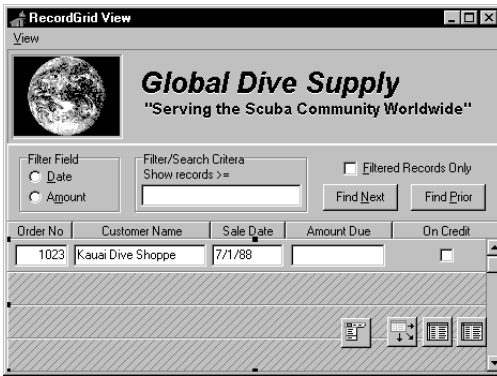
Creating a grid that contains other data-aware controls

A *TDBCtrGrid* 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.

Figure 26.5 TDBCtrGrid at design time



The following table summarizes some of the unique properties for database control grids that you can set at design time:

Table 26.10 Selected database control grid properties

Property	Purpose
AllowDelete	<i>True</i> (default): Permits record deletion. <i>false</i> : Prevents record deletion.
AllowInsert	<i>True</i> (default): Permits record insertion. <i>False</i> : Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	<i>goVertical</i> (default): Display records from top to bottom. <i>goHorizontal</i> : 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.

Table 26.10 Selected database control grid properties (continued)

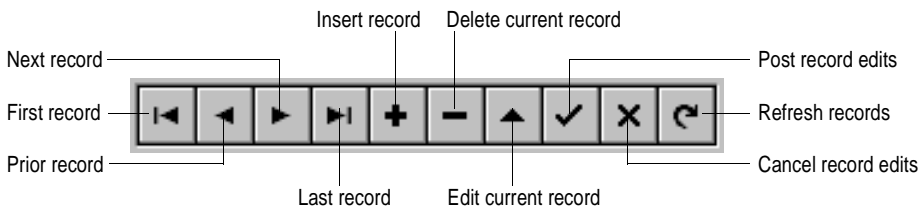
Property	Purpose
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	<i>True</i> (default): Displays a focus rectangle around the current record's panel at runtime. <i>False</i> : Does not display a focus rectangle.

For more information about database control grid properties and methods, see the online *VCL Reference*.

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.

Figure 26.6 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.

Figure 26.6 Buttons on the *TDBNavigator* control


The following table describes the buttons on the navigator.

Table 26.11 *TDBNavigator* buttons

Button	Purpose
First	Calls the dataset's <i>First</i> method to set the current record to the first record.
Prior	Calls the dataset's <i>Prior</i> method to set the current record to the previous record.
Next	Calls the dataset's <i>Next</i> method to set the current record to the next record.
Last	Calls the dataset's <i>Last</i> method to set the current record to the last record.
Insert	Calls the dataset's <i>Insert</i> method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the <i>ConfirmDelete</i> property is <i>True</i> it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.

Table 26.11 TDBNavigator buttons (continued)

Button	Purpose
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.

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, 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, double-click the *VisibleButtons* property. 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, indicating that you can collapse the list of properties by double-clicking the *VisibleButtons* property.

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.

For more information about buttons and the methods they call, see the online *VCL Reference*.

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 *OnEnter* event handler:

```

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;

```

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:

```

procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;

```


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” on page 27-2.

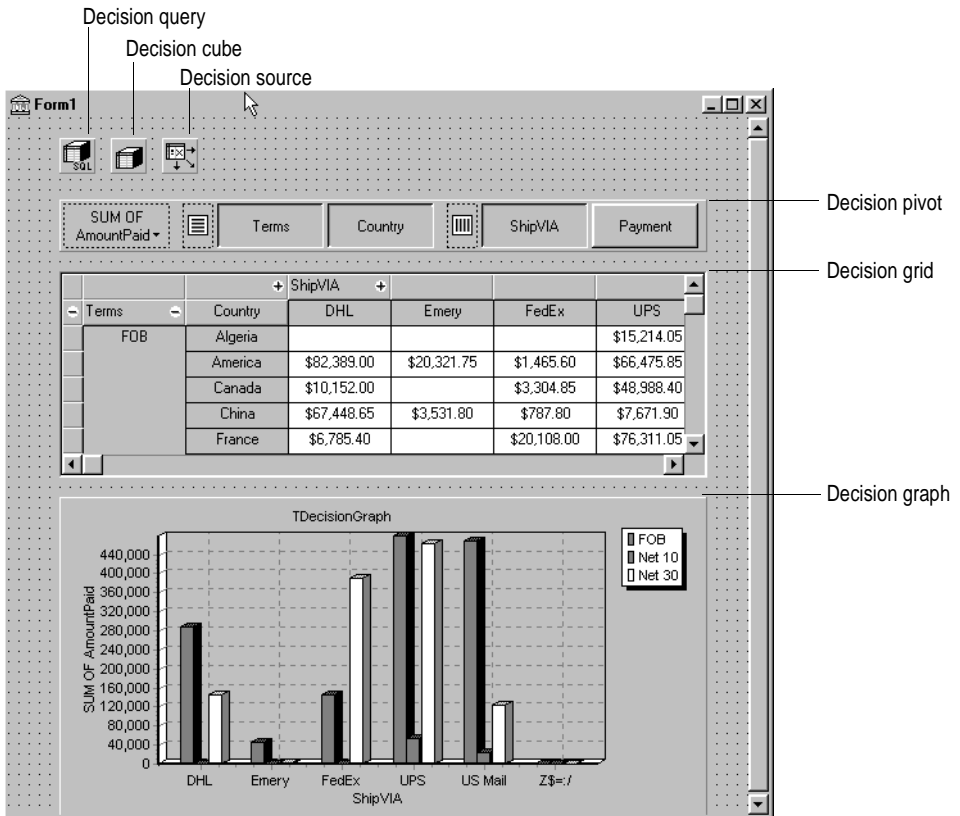
Overview

The decision support components appear on the Decision Cube page of the component palette:

- The decision cube, *TDecisionCube*, is a multidimensional data store.
- The decision source, *TDecisionSource*, defines the current pivot state of a decision grid or a decision graph.
- The decision query, *TDecisionQuery*, is a specialized form of *TQuery* used to define the data in a decision cube.
- The decision pivot, *TDecisionPivot*, lets you open or close decision cube dimensions, or fields, by pressing buttons.
- The decision grid, *TDecisionGrid*, displays single- and multidimensional data in table form.
- The decision graph, *TDecisionGraph*, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified.

Figure 27.1 shows all the decision support components placed on a form at design time.

Figure 27.1 Decision support components 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.

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 Figure 27.2 shows the amount paid for in each way.

Figure 27.2 One-dimensional crosstab

	AmEx	Cash	Check	COD	Credit	MC
	\$134,753.40	\$164,003.65	\$270,492.15	\$33,776.55	\$1,332,430.25	\$250,163.25

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 Figure 27.3.

Figure 27.3 Three-dimensional crosstab

Terms	Country	Check	COD	Credit	MC
FOB	Algeria	\$2,577.85		\$1,400.00	\$13,814.05
	America			\$356,816.20	\$20,881.35
	Canada			\$24,485.00	\$3,304.85
	China	\$61,936.90		\$6,641.55	

Guidelines for using decision support components

The decision support components listed on page 27-1 can be used together to present multidimensional data as tables and graphs. More than one table 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, follow these steps:

- 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” on page 27-6) 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" on page 27-9.
 - 4 Add one or more decision grids and graphs, bound to the decision source. For details, see "Creating and using decision grids" on page 27-10 and "Creating and using decision graphs" on page 27-13.
 - 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" on page 27-6.
 - 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" on page 27-11 and "Using decision graphs" on page 27-13 for instructions and suggestions.

For an illustration of all decision support components on a form, see Figure 27.1 on page 27-2.

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*. You can also use an ordinary *TQuery* or other dataset 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.

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” on page 27-4 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.

Using the Decision Query editor

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 the next section, “Using decision cubes,” “Using decision sources” on page 27-9, and “Using decision pivots” on page 27-9.

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 than the SQL property.

Decision query properties

The decision query has no properties than are not inherited from other components. Important inherited properties are *Active*, described in online Help and the *Visual Component Library Reference* under *TDataSet*, and *SQL*, described under *TQuery*. Queries are described in more detail in Chapter 21, “Working with queries.”

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.

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” on page 27-19.
- 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” on page 27-19.

Viewing and changing design options

To determine how much information appears during 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.

Properties and events

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 Figures 27.1, 27.2, and 27.3.

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 on page 27-2. Figure 27.1 on page 27-2 shows a decision grid on a form at design time.

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” on page 27-3.
- 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” on page 27-11.

To add a graph to the form, follow the instructions in “Creating decision graphs” on page 27-13.

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” on page 27-12.

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” on page 27-12 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” on page 27-12 for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See “Using decision pivots” on page 27-9 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” on page 27-9.

Decision grid properties

The decision grid component, *TDecisionGrid*, displays data from the *TDecisionCube* 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 page 27-2. For illustrations of decision graphs at design time, see Figure 27.1 on page 27-2 and Figure 27.4 on page 27-14.

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” on page 27-3.
- 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” on page 27-15.

For a description of what appears in the decision graph and how to use it, see the next section, “Using decision graphs.”

To add a decision grid—or crosstab table—to the form, follow the instructions in “Creating and using decision grids” on page 27-10.

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*).

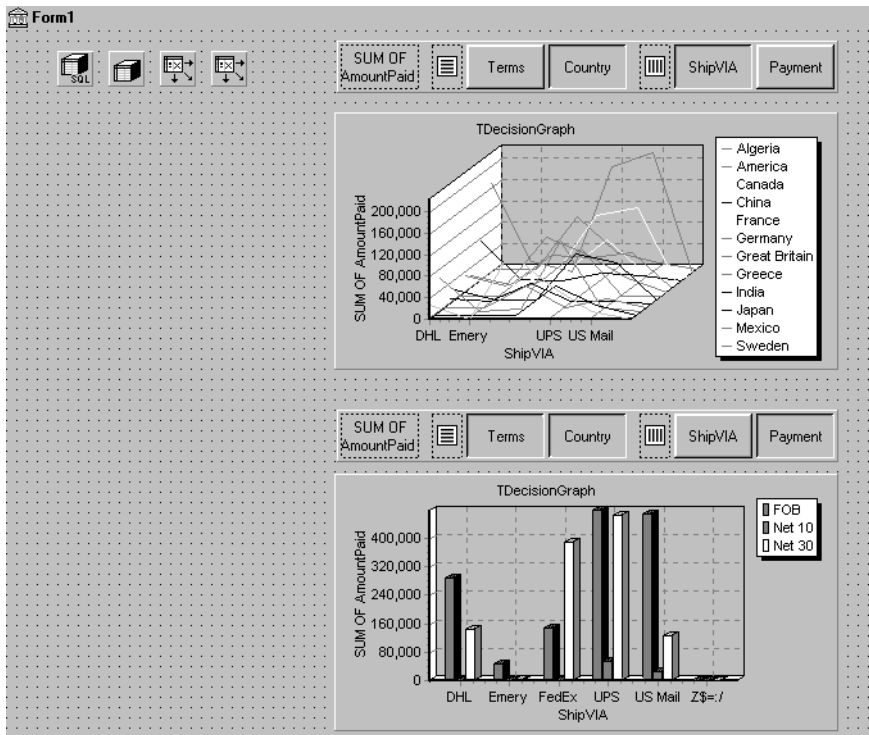
Graphed data comes from a specially formatted dataset such as *TDecisionQuery*. For an overview of how the decision support components handle and arrange this data, see page 27-1.

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 table 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 Figure 27.4 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.

Figure 27.4 Decision graphs bound to different decision sources



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” on page 27-15.

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.

For a description of the other Chart page tabs, search for the following topic in online Help: “Chart page (Chart Editing dialog box).”

- 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” on page 27-16.

To customize individual series, follow the instructions in “Customizing decision graph series” on page 27-17.

For a description of each Series page tab, search for the following topic in online Help: “Series page (Chart Editing dialog box).”

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” on page 27-16.

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 the default decision graph type” on page 27-16 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, discussed earlier in this chapter, are summarized below.

Decision pivots at runtime

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 on page 27-7.
- 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 ten times. 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.

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, look up *TDecisionCube* in the online Help.

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.

Writing distributed applications

The chapters in “Writing distributed applications” present concepts and skills necessary for building applications that are distributed over a local area network or over the internet.

Note The components described in Chapter 28 are available with the Enterprise edition of Delphi. The Web server application components discussed in Chapter 29 and the socket components discussed in Chapter 30 are available with the Professional and Enterprise editions.

Writing CORBA applications

Delphi provides wizards and classes to make it easy to create distributed applications based on the Common Object Request Broker Architecture (CORBA). CORBA is a specification adopted by the Object Management Group (OMG) to address the complexity of developing distributed object applications.

As its name implies, CORBA provides an object-oriented approach to writing distributed applications. This is in contrast to a message-oriented approach such as the one described for HTTP applications in Chapter 29, “Creating Internet server applications”. Under CORBA, server applications implement objects that can be used remotely by client applications, through well-defined interfaces.

Note COM provides another object-oriented approach to distributed applications. For more information about COM, see Chapter 44, “Overview of COM technologies”. Unlike COM, however, CORBA is a standard that applies to platforms other than Windows. This means you can write CORBA clients or servers using Delphi that can communicate with CORBA-enabled applications running on other platforms.

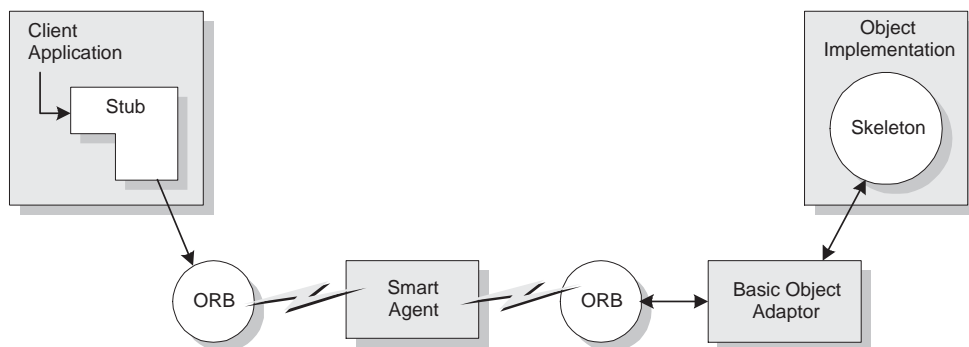
The CORBA specification defines how client applications communicate with objects that are implemented on a server. This communication is handled by an Object Request Broker (ORB). Delphi’s CORBA support is based on the VisiBroker for C++ ORB (Version 3.3.2) with a special wrapper (orbpas.dll) that exposes a subset of the ORB functionality to Delphi applications.

In addition to the basic ORB technology, which enables clients to communicate with objects on server machines, the CORBA standard defines a number of standard services. Because these services use well-defined interfaces, developers can write clients that use these services even if the servers are written by different vendors.

Overview of a CORBA application

If you are already doing object-oriented programming, CORBA makes writing distributed applications easy, because it lets you use remote objects almost as if they were local. This is because the design of a CORBA application is much like any other object-oriented application, except that it includes an additional layer for handling network communication when an object resides on a different machine. This additional layer is handled by special objects called stubs and skeletons.

Figure 28.1 The structure of a CORBA application



On CORBA clients, the stub acts as a proxy for an object that may be implemented by the same process, another process, or on another (server) machine. The client interacts with the stub as if it were any other object that implements an interface. For more information about using interfaces, see “Using interfaces” on page 3-14.

However, unlike most objects that implement interfaces, the stub handles interface calls by calling into the ORB software that is installed on the client machine. The VisiBroker ORB uses a Smart Agent (osagent) that is running somewhere on the local area network. The Smart Agent is a dynamic, distributed directory service that locates an available server which provides the real object implementation.

On the CORBA server, the ORB software passes interface calls to an automatically-generated skeleton. The skeleton communicates with the ORB software through the Basic Object Adaptor (BOA). Using the BOA, the skeleton registers the object with the Smart Agent, indicates the scope of the object (whether it can be used on remote machines), and indicates when objects are instantiated and ready to respond to clients.

Understanding stubs and skeletons

Stubs and skeletons provide the mechanism that allows CORBA applications to marshal interface calls. Marshaling

- Takes an interface pointer in the server’s process and makes the 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 caller pushes arguments onto the stack and makes a function call through the interface pointer. If the object is not in the same process space as the code that calls its interface, the call gets passed to a stub which is in the same process space. The stub writes the arguments into a marshaling buffer and transmits the call in a structure to the remote object. The server skeleton unpacks this structure, pushes the arguments onto the stack, and calls the object's implementation. In essence, the skeleton recreates the client's call in its own address space.

Stubs and skeletons are created for you automatically when you define the object's interface. Their definitions are created in the `_TLB` unit that is created when you define the interface. You can view this unit by selecting it in your implementation unit's `uses` clause and typing *Control-Enter*. For more information about defining the object's interface, see "Defining object interfaces" on page 28-5.

Using Smart Agents

The Smart Agent (osagent) is a dynamic, distributed directory service that locates an available server that implements an object. If there are multiple servers to choose from, the Smart Agent provides load balancing. It also protects against server failures by attempting to restart the server when a connection fails, or, if necessary, locating a server on another host.

A Smart Agent must be started on at least one host in your local network, where local network refers to a network within which a broadcast message can be sent. The ORB locates a Smart Agent by using a broadcast message. If the network includes multiple Smart Agents, the ORB uses the first one that responds. Once the Smart Agent is located, the ORB uses a point-to-point UDP protocol to communicate with the Smart Agent. The UDP protocol consumes fewer network resources than a TCP connection.

When a network includes multiple Smart Agents, each Smart Agent recognizes a subset of the objects available, and communicates with other Smart Agents to locate objects it can't recognize directly. If one Smart Agent terminates unexpectedly, the objects it keeps track of are automatically re-registered with another available Smart Agent.

For details about configuring and using Smart Agents on your local networks, see "Configuring Smart Agents" on page 28-17.

Activating server applications

When the server application starts, it informs the ORB (through the Basic Object Adaptor) of the objects that can accept client calls. This code to initialize the ORB and inform it that the server is up and ready is added to your application automatically by the wizard you use to start your CORBA server application.

Typically, CORBA server applications are started manually. However, you can use the Object Activation Daemon (OAD) to start your servers or instantiate their objects only when clients need to use them.

To use the OAD, you must register your objects with it. When you register your objects with the OAD, it stores the association between your objects and the server application that implements them in a database called the Implementation Repository.

Once there is an entry for your object in the Implementation Repository, the OAD simulates your application to the ORB. When a client requests the object, the ORB contacts the OAD as if it were the server application. The OAD then forwards the client request to the real server, launching the application if necessary.

For details about registering your objects with the OAD, see “Registering interfaces with the Object Activation Daemon” on page 28-9.

Binding interface calls dynamically

Typically, CORBA clients use static binding when calling the interfaces of objects on the server. This approach has many advantages, including faster performance and compile-time type checking. However, there are times when you can't know until runtime what interface you want to use. For these cases, Delphi lets you bind to interfaces dynamically at runtime.

Before you can take advantage of dynamic binding, you must register your interfaces with the Interface Repository using the `idl2ir` utility. “Registering interfaces with the Interface Repository” on page 28-8 describes how to do this.

For details on how to use dynamic binding in your CORBA client applications, see “Using the dynamic invocation interface” on page 28-12.

Writing CORBA servers

Two wizards on the Multi Tier page of the New Items dialog let you create CORBA servers:

- The CORBA Data Module wizard lets you create a CORBA server for a multi-tiered database application.
- The CORBA Object wizard lets you create an arbitrary CORBA server.

In addition, you can easily convert an existing Automation server to a CORBA server by right-clicking and choosing `Expose As CORBA Object`. When you expose an Automation server as a CORBA object, you create a single application that can service both COM clients and CORBA clients simultaneously.

Using the CORBA wizards

To start the wizard, choose `File | New` to display the New Items dialog. Select the Multi Tier page, and double-click the appropriate wizard.

You must supply a class name for your CORBA object. This is the base name of a descendant of `TCorbaDataModule` or `TCorbaImplementation` that your application

creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyObject*, the wizard creates a new unit declaring *TMyObject* which implements the *IMyObject* interface.

The wizard lets you specify how you want your server application to create instances of this object. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the object that handles all client requests. This is the model used in traditional CORBA development. Because the single object instance is shared by all clients, it must not rely on any persistent state information such as property settings.
- When you choose instance-per-client, a new object instance is created for each client connection. This instance persists until a specified timeout period elapses with no messages from the client. This model lets you use persistent state information, because separate clients can't interfere with each other's property settings. However, client applications must call the server often enough so that the server object does not time out.

Note The instance-per-client model is not typical of most CORBA development, but works, rather, like the COM model, where the lifetime of a server object is governed by client usage. This model allows Delphi to create servers that act as CORBA and COM servers simultaneously.

In addition to the instancing model, you must specify the threading model. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each object instance is guaranteed to receive only one client request at a time. You can safely access your object's instance data (properties or fields). However, you must guard against thread conflicts when you use global variables or objects.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared object instance, because you must protect against thread conflicts for all data and objects contained in your application.

Defining object interfaces

With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the Type Library editor and Delphi automatically updates the appropriate source files. For more information on defining interfaces using the Type Library editor, see Chapter 50, "Working with type libraries."

The Type Library editor is also used for defining COM-based type libraries. Because of this, it includes many options and controls that are not relevant to CORBA applications. If you try to use these options, (for example, if you try to specify a

version number or help file), your settings are ignored. If you create a COM Automation server which you then expose as a CORBA server, these settings apply to your server in its role as an Automation server.

In the Type Library editor, you can define your interface using Object Pascal syntax or the Microsoft IDL that is used for COM objects. Specify which language you want to use when defining your interfaces on the Type Library page of the Environment Options dialog. If you choose to use IDL, be aware that the Microsoft IDL differs slightly from the CORBA IDL. When defining your interface, you are limited to the types listed in the following table:

Table 28.1 Types allowed in a CORBA interface

Type	Details
ShortInt	8-bit signed integer
Byte	8-bit unsigned integer
SmallInt	16-bit signed integer
Word	16-bit unsigned integer
Longint, Integer	32-bit signed integer
Cardinal	32-bit unsigned integer
Single	4-byte floating point value
Double	8-byte floating point value
TDateTime	Passed as a Double value
PWideChar	Unicode string
String, PChar	Strings must be cast to a PChar
VARIANT	Passed as a CORBA Any. This is the only way to pass an Array or Currency value.
Boolean	passed as a CORBA_Boolean (Byte)
Object Reference or interface	Passed as a CORBA interface
Enumerated types	Passed as an Integer

Note Instead of using the Type Library editor, you can add to your interface by right-clicking in the code editor and choosing Add To Interface. However, you will need to use the Type Library editor to save an .IDL file for your interface.

You can't add properties that use parameters (although you can add get and set methods for such properties). Some types (such as arrays, Int64 values, or Currency types) must be specified as Variants. Records are not supported in the Client/Server version.

Your interface definition is reflected in the automatically generated stub-and-skeleton unit. This unit is updated when you choose Refresh in the type library editor or when you use the Add To Interface command. This automatically generated unit is added to the uses clause of your implementation unit. Do not edit the stub-and-skeleton unit.

In addition to the stub-and-skeleton unit, editing the interface updates your server implementation unit by adding declarations for your interface members and providing empty implementations for the methods. You can then edit this

implementation unit to provide meaningful code for the body of each new interface method.

Note You can save a CORBA .IDL file for your interface by clicking the Export button while in the Type Library editor. Specify that the .IDL file should use CORBA IDL, not Microsoft IDL. Use this .IDL file for registering your interface or for generating stubs and skeletons for other languages.

Automatically generated code

When you define CORBA object interfaces, two unit files are automatically updated to reflect your interface definitions.

The first of these is the stub-and-skeleton unit. It has a name of the form `MyInterface_TLB.pas`. While this unit defines the stub class that is only used by client applications, it also contains the declaration for your interface types and your skeleton classes. You should not edit this file directly. However, this unit is automatically added to the **uses** clause of your implementation unit.

The stub-and-skeleton unit defines a skeleton object for every interface supported by your CORBA server. The skeleton object is a descendant of *TCorbaSkeleton*, and handles the details of marshaling interface calls. It does not implement the interfaces you define. Instead, its constructor takes an interface instance which it uses to handle all interface calls.

The second updated file is the implementation unit. By default, it has a name of the form `unit1.pas`, although you will probably want to change this to a more meaningful name. This is the file that you edit.

For each CORBA interface you define, an implementation class definition is automatically added to your implementation unit. The implementation class name is based on the interface name. For example, if the interface is *IMyInterface*, the implementation class is named *TMyInterface*. You will find code added to this class's implementation for every method you add to the interface. You must fill in the body of these methods to finish the implementation class.

In addition, you may notice that some code is added to the initialization section of your implementation unit. This code creates a *TCorbaFactory* object for each object interface you expose to CORBA clients. When clients call your CORBA server, the CORBA factory object creates or locates an instance of your implementation class and passes it as an interface to the constructor for the corresponding skeleton class.

Note The use of factory objects that indirectly create your CORBA server objects is not typical of most CORBA development. Instead, it follows the model used by COM server applications, and enables Delphi to build servers that act as COM and CORBA servers simultaneously. Factories allow CORBA servers to implement the instance-per-client model. Clients of CORBA servers built with Delphi use the *CorbaFactoryCreateStub* function, which handles the details of instructing the factory on the CORBA server to create a CORBA object.

Registering server interfaces

While it is not necessary to register your server interfaces if you are only using static binding of client calls into your server objects, registering your interfaces is recommended. There are two utilities with which you can register your interfaces:

- **The Interface Repository.** By registering with the Interface Repository, clients can take advantage of dynamic binding. This allows your server to respond to clients that are not written in Delphi if they use the dynamic invocation interface (DII). For more information about using DII, see “Using the dynamic invocation interface” on page 28-12. Registering with the Interface Repository is also a convenient way to allow other developers to view your interfaces when they write client applications.
- **The Object Activation Daemon.** By registering with the Object Activation Daemon (OAD), your server need not be launched or your objects instantiated until they are needed by clients. This conserves resources on your server system.

Registering interfaces with the Interface Repository

You can create an Interface Repository for your interfaces by running the Interface Repository server. First, you must save the .IDL file for your interface. To do this, choose View | Type Library, and then, in the Type Library Editor, click the Export button to export your interface as a CORBA .IDL file.

Once you have an .IDL file for your interface, you can run the Interface Repository Server using the following syntax:

```
irep [-console] IRname [file.idl]
```

The irep arguments are described in the following table:

Table 28.2 irep arguments

Argument	Description
-console	Starts the Interface Repository server as a console application. By default, the Interface Repository server runs as a Windows application.
IRname	The name of the Interface Repository. While the server is running, clients use this name to bind to the Interface Repository so that they can obtain interface information for DII or so that they can register additional interfaces.
file.idl	An .IDL file that describes the initial contents of the Interface Repository. If you do not specify a file name, the Interface Repository starts out empty. You can subsequently add interfaces using the menus of the Interface Repository server, or using the idl2ir utility.

Once the Interface Repository server is running, you can add additional interfaces by choosing File | Load and specifying a new .IDL file. However, if the new .IDL file contains any entries that match an existing .IDL entry, the new .IDL file is rejected.

At any point, you can save the current contents of the Interface Repository to an .IDL file by choosing File | Save or File | Save As. This way, after you exit the Interface

Repository, you can restart it later with the saved file so that you don't need to reimport all changes to the initial .IDL file.

You can also register additional interfaces with the Interface Repository using the `idl2ir` utility. While the Interface Repository server is running, start the `idl2ir` utility using the following syntax:

```
idl2ir [-ir IRname] {-replace} file.idl
```

The `idl2ir` arguments are described in the following table:

Table 28.3 `idl2ir` arguments

Argument	Description
-ir IRname	Directs the utility to bind to the interface repository instance named IRname. If this argument is not specified, <code>idl2ir</code> binds to any interface repository returned by the smart agent.
-replace	Directs the utility to replace interface repository items with matching items in <code>file.idl</code> . If <code>-replace</code> is not specified, the entire interface is added to the repository, unless there are matching items, in which case the utility rejects the entire .IDL file. If <code>-replace</code> is specified, non-matching items are rejected.
file.idl	Specifies the .IDL file that contains the updates for the Interface Repository.

Entries in an interface repository can't be removed while the Interface Repository server is running. To remove an item, you must shut down the Interface Repository server, generate a new .IDL file, and then start the Interface Repository server, specifying the updated .IDL file.

Registering interfaces with the Object Activation Daemon

Before you can register an interface with the Object Activation Daemon (OAD), the OAD command-line program must be running on at least one machine on your local network. Start the OAD using the following syntax:

```
oad [options]
```

The OAD utility accepts the following command line arguments:

Table 28.4 OAD arguments

Argument	Description
-v	Turns on verbose mode.
-f	Stipulates that the process should not fail if another OAD is running on this host.
-t<n>	Specifies the number of seconds the OAD will wait for a spawned server to activate the requested object. The default time-out is 20 seconds. Setting this value to 0 causes the OAD to wait indefinitely. If the spawned server process does not activate the requested object within the time-out period, the OAD terminates the server process and returns an exception.
-C	Allows the OAD to run in console mode if it has been installed as an NT service.
-k	Stipulates that an object's child process should be killed once all of its objects are unregistered with the OAD.
-?	Describes these arguments.

Once the OAD is running, you can register your object interfaces using the command-line program `oadutil`. First, you must export the .IDL file for your interfaces. To do this, click the **Export** button in the Type Library editor and save the interface definition as a CORBA .IDL file.

Next, register interfaces using the `oadutil` program with the following syntax:

```
oadutil reg [options]
```

The following arguments are available when registering interfaces using `oadutil`:

Table 28.5 `oadutil reg` arguments

Arguments	Description
<code>-i <interface name></code>	Specifies a particular IDL interface name. You must specify the interface to register using either this or the <code>-r</code> option.
<code>-r <repository id></code>	Specifies a particular interface by its repository id. The repository id is a unique identifier associated with the interface. You must specify the interface to register using this or the <code>-i</code> option.
<code>-o <object name></code>	Specifies the name of the object that supports the interface. This option is required.
<code>-cpp <file name></code>	Specifies the name of your server executable. This option is required.
<code>-host <host name></code>	Specifies a remote host where the OAD is running. (optional)
<code>-verbose</code>	Starts the utility in verbose mode. Messages are sent to stdout. (optional)
<code>-d <reference data></code>	Specifies reference data that is passed to the server application on activation. (optional)
<code>-a arg1</code>	Specifies command-line arguments that are passed to the server application. Multiple arguments can be passed with multiple <code>-a</code> parameters. (optional)
<code>-e env1</code>	Specifies environment variables that are passed to the server application. Multiple arguments are passed with multiple <code>-e</code> parameters. (optional)

For example, the following line registers an interface based on its repository id:

```
oadutil reg -r IDL:MyServer/MyObjectFactory:1.0 -o TMyObjectFactory -cpp MyServer.exe -p unshared
```

Note You can obtain the repository ID for your interface by looking at the code added to the initialization section of your implementation unit. It appears as the third argument of the call to `TCorbaFactory.Create`.

When an interface becomes unavailable, you must unregister it. Once an object is unregistered, it can no longer be automatically activated by the OAD if a client requests the object. Only objects that have been previously registered using `oadutil reg` can be unregistered.

To unregister interfaces, use the following syntax:

```
oadutil unreg [options]
```


The following arguments are available when unregistering interfaces using `oadutil`:

Table 28.6 `oadutil` unreg arguments

Argument	Description
<code>-i <interface name></code>	Specifies a particular IDL interface name. You must specify the interface to unregister using either this or the <code>-r</code> option.
<code>-r <repository id></code>	Specifies a particular interface by its repository id. The repository id is a unique identifier associated with the interface when it is registered with the Interface Repository. You must specify the interface to unregister using this or the <code>-i</code> option.
<code>-o <object name></code>	Specifies the name of the object that supports the interface. If you do not specify the object name, all objects that support the specified interface are unregistered.
<code>-host <host name></code>	Specifies a remote host where the OAD is running. (optional)
<code>-verbose</code>	Starts the utility in verbose mode. Messages are sent to stdout. (optional)
<code>-version</code>	Displays the version number for <code>oadutil</code> .

Writing CORBA clients

When you write a CORBA client, the first step is to ensure that the client application can talk to the ORB software on the client machine. To do this, simply add `CorbaInit` to the **uses** clause of your unit file. Next, proceed with writing your application in the same way you write any other application in Delphi. However, when you want to use objects that are defined in the server application, you do not work directly with an object instance. Instead, you obtain an interface for the object and work with that. You can obtain the interface in one of two ways, depending on whether you want to use static or dynamic binding.

To use static binding, you must add a stub-and-skeleton unit to your client application. The stub-and-skeleton unit is created automatically when you save the server interface. Using static binding is faster than using dynamic binding, and provides additional benefits such as compile-time type checking and code-completion.

However, there are times when you do not know until runtime what object or interface you want to use. For these cases, you can use dynamic binding. Dynamic binding does not require a stub unit, but it does require that all remote object interfaces you use are registered with an Interface Repository running on the local network.

Tip You may want to use dynamic binding when writing CORBA clients for servers that are not written in Delphi. This way, you do not need to write your own stub class for marshaling interface calls.

Using stubs

Stub classes are generated automatically when you define a CORBA interface. They are defined in a stub-and-skeleton unit, which has a name of the form `BaseName_TLB` (in a file with a name of the form `BaseName_TLB.pas`).

When writing a CORBA client, you do not edit the code in the stub-and-skeleton unit. Add the stub-and-skeleton unit to the uses clause of the unit which needs an interface for an object on the CORBA server. For each server object, the stub-and-skeleton unit contains an interface definition and a class definition for a corresponding stub class. For example, if the server defines an object class *TServerObj*, the stub-and-skeleton unit includes a definition for the interface *IServerObj*, and for a stub class *TServerObjStub*. The stub class is a descendant of *TCorbaDispatchStub*, and implements its corresponding interface by marshaling calls to the CORBA server. In addition to the stub class, the stub-and-skeleton unit defines a stub factory class for each interface. This stub factory class is never instantiated: it defines a single class method.

In your client application, you do not directly create instances of the stub class when you need an interface for the object on the CORBA server. Instead, call the class method *CreateInstance* of the stub factory class. This method takes one argument, an optional instance name, and returns an interface to the object instance on the server. For example:

```
var
  ObjInterface : IServerObj;
begin
  ObjInterface := TServerObjFactory.CreateInstance('');
  ...
end;
```

When you call *CreateInstance*, it

- 1 Obtains an interface instance from the ORB.
- 2 Uses that interface to create an instance of the stub class.
- 3 Returns the resulting interface.

Note If you are writing a client for a CORBA server that was not written using Delphi, you must write your own descendant of *TCorbaStub* to provide marshaling support for your client. You must then register this stub class with the global *CORBASTubManager*. Finally, to instantiate the stub class and get the server interface, you can call the global *BindStub* procedure to obtain an interface which you then pass to the CORBA stub manager's *CreateStub* method.

Using the dynamic invocation interface

The dynamic invocation interface (DII) allows client applications to call server objects without using a stub class that explicitly marshals interface calls. Because DII must encode all type information before the client sends a request and then decode that information on the server, it is slower than using a stub class.

Before you can use DII, the server interfaces must be registered with an Interface Repository that is running on the local network. For more information about registering interfaces with the Interface Repository, see “Registering interfaces with the Interface Repository” on page 28-8.

To use DII in a client application, obtain a server interface and assign it to a variable of type *TAny*. *TAny* is a special, CORBA-specific Variant. Then call the methods of the interface using the *TAny* variable as if it were an interface instance. The compiler handles the details of turning your calls into DII requests.

Obtaining the interface

To get an interface for making late-bound DII calls, use the global *CorbaBind* function. *CorbaBind* takes either the Repository ID of the server object or an interface type. It uses this information to request an interface from the ORB, and uses that to create a stub object.

Note Before calling *CorbaBind*, the association between the interface type and its Repository ID must be registered with the global *CorbaInterfaceIDManager*.

If your client application has a registered stub class for the interface type, *CorbaBind* creates a stub of that class. In this case, the interface returned by *CorbaBind* can be used for both early binding (by casting with the **as** operator) or late (DII) binding. If there is no registered stub class for the interface type, *CorbaBind* returns the interface to a generic stub object. A generic stub object can only be used for late (DII) calls.

To use the interface returned by *CorbaBind* for DII calls, assign it to a variable of type *TAny*:

```
var
  IntToCall: TAny;
begin
  IntToCall := CorbaBind('IDL:MyServer/MyServerObject:1.0');
  ...
```

Calling interfaces with DII

Once an interface has been assigned to a variable of type *TAny*, calling it using DII simply involves using the variable as if it were an interface:

```
var
  HR, Emp, Payroll, Salary: TAny;
begin
  HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
  Emp := HR.LookupEmployee(Edit1.Text);
  Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

When using DII, all interface methods are case sensitive. Unlike when making statically-bound calls, you must be sure that method names match the case used in the interface definition.

When calling an interface using DII, every parameter is treated as a value of type *TAny*. This is because *TAny* values carry their type information with them. This type information allows the server to interpret type information when it receives the call.

Because the parameters are always treated as *TAny* values, you do not need to explicitly convert to the appropriate parameter type. For example, in the previous example, you could pass a string instead of a floating-point value for the last parameter in the call to *SetEmployeeSalary*:

```
Payroll.SetEmployeeSalary(Emp, Edit2.Text);
```

You can always pass simple types directly as parameters, and the compiler converts them to *TAny* values. For structured types, you must use the conversion methods of the global ORB variable to create an appropriate *TAny* type. Table 28.7 indicates the method to use for creating different structured types:

Table 28.7 ORB methods for creating structured TAny values

Structured type	helper function
record	MakeStructure
array (fixed length)	MakeArray
dynamic array (sequence)	MakeSequence

When using these helper functions, you must specify the type code that describes the type of record, array, or sequence you want to create. You can obtain this type dynamically from a Repository ID using the ORB's *FindTypeCode* method:

```
var
  HR, Name, Emp, Payroll, Salary: TAny;
begin
  with ORB do
    begin
      HR := Bind('IDL:CompanyInfo/HR:1.0');
      Name := MakeStructure(FindTypeCode('IDL:CompanyInfo/EmployeeName:1.0',
        [Edit1.Text, Edit2.Text]));
      Emp := HR.LookupEmployee(Name);
      Payroll := Bind('IDL:CompanyInfo/Payroll:1.0');
    end;
    Salary := Payroll.GetEmployeeSalary(Emp);
    Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit3.Text) / 100));
  end;
```

Customizing CORBA applications

Two global functions, *ORB* and *BOA*, allow you to customize the way your application interacts with the CORBA software running on your network.

Client applications use the value returned by *ORB* to configure the ORB software, disconnect from the server, bind to interfaces, and obtain string representations for objects so that they can display object names in the user interface.

Server applications use the value returned by *BOA* to configure the BOA software, expose or hide objects, and retrieve custom information assigned to an object by client applications.

Displaying objects in the user interface

When writing a CORBA client application, you may wish to present users with the names of available CORBA server objects. To do this, you must convert your object interface to a string. The *ObjectToString* method of the global *ORB* variable performs this conversion. For example, the following code displays the names of three objects in a list box, given interface instances for their corresponding stub objects.

```
var
  Dept1, Dept2, Dept3: IDepartment;
begin
  Dept1 := TDepartmentFactory.CreateInstance('Sales');
  Dept1.SetDepartmentCode(120);
  Dept2 := TDepartmentFactory.CreateInstance('Marketing');
  Dept2.SetDepartmentCode(98);
  Dept3 := TSecondFactory.CreateInstance('Payroll');
  Dept3.SetDepartmentCode(49);
  ListBox1.Items.Add(ORB.ObjectToString(Dept1));
  ListBox1.Items.Add(ORB.ObjectToString(Dept2));
  ListBox1.Items.Add(ORB.ObjectToString(Dept3));
end;
```

The advantage of letting the ORB create strings for your objects is that you can use the *StringToObject* method to reverse this procedure:

```
var
  Dept: IDepartment;
begin
  Dept := ORB.StringToObject(ListBox1.Items[ListBox1.ItemIndex]);
  ... { do something with the selected department }
```

Exposing and hiding CORBA objects

When a CORBA server application creates an object instance, it can make that object available to clients by calling the *ObjIsReady* method of the global variable returned by *BOA*.

Any object exposed using *ObjIsReady* can be hidden by the server application. To hide an object, call the BOA's *Deactivate* method.

If the server deactivates an object, this can invalidate an object interface held by a client application. Client applications can detect this situation by calling the stub object's *NonExistent* method. *NonExistent* returns *True* when the server object has been deactivated and *False* if the server object is still available.

Passing client information to server objects

Stub objects in CORBA clients can send identifying information to the associated server object using a *TCorbaPrincipal*. A *TCorbaPrincipal* is an array of bytes that represents information about the client application. Stub objects set this value using their *SetPrincipal* method.

Once the CORBA client has written principal data to the server object instance, the server object can access this information using the BOA's *GetPrincipal* method.

Because *TCorbaPrincipal* is an array of bytes, it can represent any information that the developer finds useful to send. For example, clients with special privileges can send a key value that the server checks before making some methods available.

Deploying CORBA applications

Once you have created client or server applications and thoroughly tested them, you are ready to deploy client applications to end users' desktops and server applications on server-class machines. The following list describes the files that must be installed (in addition to your client or server application) when you deploy your CORBA application:

- The ORB libraries must be installed on every client and server machine. These libraries are found in the Bin subdirectory of the directory where you installed VisiBroker.
- If your client uses the dynamic invocation interface (DII), you must run an Interface Repository server on at least one host on the local network, where local network refers to a network within which a broadcast message can be sent. See "Registering interfaces with the Interface Repository" on page 28-8 for details on how to run the Interface Repository server and register the appropriate interfaces.
- If your server should be started on demand, the Object Activation Daemon (OAD) must be running on at least one host on the local network. See "Registering interfaces with the Object Activation Daemon" on page 28-9 for details on how to run the OAD and register the appropriate interfaces.
- A Smart Agent (osagent) must be installed on at least one host on the local network. You may want to deploy the Smart Agent to more than one machine.

In addition, you may need to set the following environment variables when you deploy your CORBA application.

Table 28.8 CORBA environment variables

Variable	Meaning
PATH	You must ensure that the directory which contains the ORB libraries is on the Path.
VBROKER_ADM	Specifies the directory that contains configuration files for the Interface Repository, Object Activation Daemon, and Smart Agent.
OSAGENT_ADDR	Specifies the IP address of the host machine whose Smart Agent should be used. If this variable is not set, the CORBA application uses the first Smart Agent that answers a broadcast message.
OSAGENT_PORT	Specifies the port which a Smart Agent uses to listen for requests.
OSAGENT_ADDR_FILE	Specifies the fully qualified path name to a file containing Smart Agent addresses on other local networks.
OSAGENT_LOCAL_FILE	Specifies the fully qualified path name to a file containing network information for a Smart Agent running on a multi-homed host.
VBROKER_IMPL_PATH	Specifies the directory for the Implementation Repository (where the OAD stores its information).
VBROKER_IMPL_NAME	Specifies the default file name for the Implementation Repository.

Note For general information about deploying applications, see Chapter 11, “Deploying applications”.

Configuring Smart Agents

When you deploy your CORBA application, there must be at least one smart agent running on the local network. By deploying more than one smart agent on a local network, you provide protection against the machine that is running the smart agent from going down.

You deploy smart agents so that they divide a local network into separate ORB domains. Conversely, you can connect smart agents on different local networks to broaden the domain of your ORB.

Starting the Smart Agent

To start the Smart Agent, run the `osagent` utility. You must run at least one Smart Agent on a host in your local network. The `osagent` utility accepts the following command line arguments:

Table 28.9 `osagent` arguments

Argument	Description
-v	Turns on verbose mode. Information and diagnostic messages are written to a log file named <code>osagent.log</code> . You can find this file in the directory specified by the <code>VBROKER_ADM</code> environment variable.
-p<n>	Specifies the UDP port that the Smart Agent uses to listen for broadcast messages.
-C	Allows the Smart Agent to run in console mode if it has been installed as an NT service.

For example, type the following command in a DOS box or choose Run from the Start button:

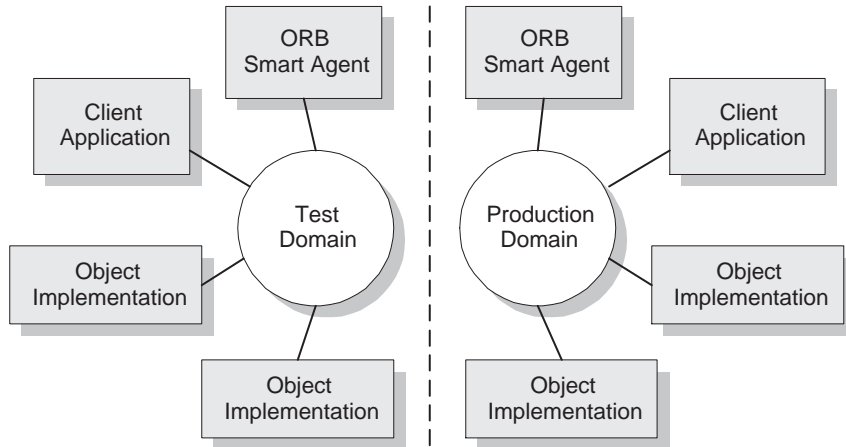
```
osagent -p 11000
```

This starts the Smart Agent so that it listens on UDP port 11000 rather than the default port (14000). Changing the port which the Smart Agent uses to listen for broadcast messages allows you to create multiple ORB domains.

Configuring ORB domains

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production version of client applications and object implementations while another domain could include test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want a dedicated ORB domain so that the testing efforts of different developers do not conflict with each other.

Figure 28.2 Separate ORB domains



You can distinguish between two or more ORB domains on the same network by using a unique UDP port number of the osagents in each domain.

The default port number (14000) is written to the Windows Registry when the ORB is installed. To override this value, set the OSAGENT_PORT environment variable to a different setting. You can further override the value specified by OSAGENT_PORT by starting the Smart Agent using the -p option.

Connecting Smart Agents on different local networks

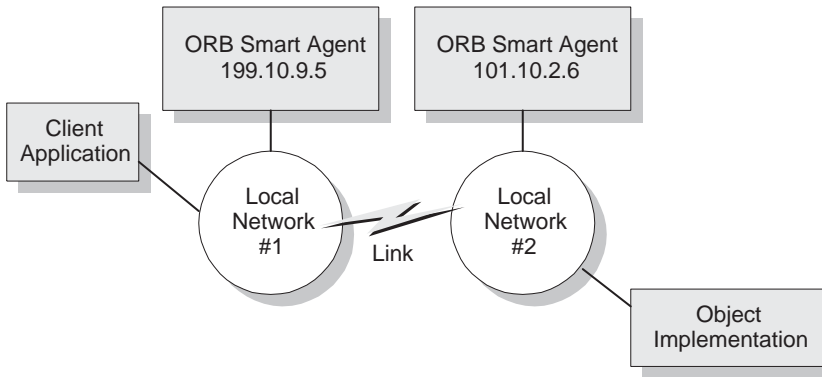
If you start multiple Smart agents on your local network, they will discover each other using UDP broadcast messages. Your local networks are configured by specifying the scope of broadcast messages using the IP subnet mask. You can allow a Smart Agent to communicate with other networks in two ways:

- Using an agentaddr file.
- Using a multi-homed host.

Using an agentaddr file

Consider the two Smart Agents depicted in the following figure. The Smart Agent on network #1 listens for broadcast messages using the IP address 199.10.9.5. The Smart Agent on network #2 listens using the IP address 101.10.2.6.

Figure 28.3 Two Smart Agents on separate local networks



The Smart Agent on network #1 can contact the Smart Agent on network #2 if it can find a file named **agentaddr** which contains the following line:

```
101.10.2.6
```

The Smart Agent looks for this file in the directory specified by the VBROKER_ADM environment variable.

Using a multi-homed host

When you start the Smart Agent on a host that has more than one IP address (known as a multi-homed host), it can provide a powerful mechanism for bridging objects located on separate local networks. All local networks to which the host is connected can communicate with a single Smart Agent, effectively bridging the local networks without an **agentaddr** file.

However, on a multi-homed host, the Smart Agent can't determine the correct subnet mask and broadcast address values. You must specify these values in a **localaddr** file. You can obtain the appropriate network interface values of the **localaddr** file by accessing the TCP/IP protocol properties from the Network Control Panel. If your host is running Windows NT, you can use the ipconfig command to get these values.

The **localaddr** file contains a line for every combination of IP address, subnet mask, and broadcast address that the Smart Agent can use. For example, the following lists the contents of a **localaddr** file for a Smart Agent that has two IP addresses:

```
216.64.15.10 255.255.255.0 216.64.15.255
214.79.98.88 255.255.255.0 214.79.98.255
```

You must also set the OSAGENT_LOCAL_FILE environment variable to the fully qualified path name of the **localaddr** file. This allows the Smart Agent to locate this file.

Creating Internet server applications

Delphi allows you to create Web server applications as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Special components on the Internet palette page make it easy to create event handlers that are associated with a specific Uniform Resource Identifier (URI) and, when processing is complete, to programmatically construct HTML documents and transfer them to the client.

Frequently, this content is drawn from databases. Internet components can be used to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

This chapter describes these Internet components, and discusses the creation of several types of Internet applications.

Note You can also use ActiveForms as Internet server applications. For more information about ActiveForms, see “Generating an ActiveX control based on a VCL form” on page 48-6.

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.

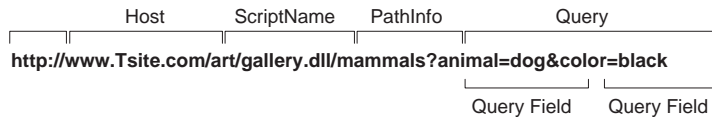
- RFC1945, “Hypertext Transfer Protocol — HTTP/1.0,” describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, www.ietf.cnri.reston.va.us

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 Figure 29.1:

Figure 29.1 Parts of a Uniform Resource Locator



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 of 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=dog&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

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 WinCGI, the server opens a file and writes out the request information. It then executes the Win-CGI program, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. The server waits while the program executes. When the program exits, the server reads the data from the content file written by the Win-CGI program.
- 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 also differs based on the type of program:

- When a Win-CGI script finishes it constructs a page of HTML, writes it to a file, writes any response information to another file, and passes the locations of both files back to the server. The server opens both files and passes the HTML page back to the client.
- 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.

Web server applications

Web server applications extend the functionality and capability of existing Web servers. The 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. Any operation that you can perform with a Delphi application can be incorporated into a Web server application.

Types of Web server applications

Using the internet components, you can create four types of Web server applications. Each type uses a type-specific descendant of *TWebApplication*, *TWebRequest*, and *TWebResponse*:

Table 29.1 Web server application components

Application Type	Application Object	Request Object	Response Object
Microsoft Server DLL (ISAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Netscape Server DLL (NSAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Console CGI application	<i>TCGIApplication</i>	<i>TCGIRequest</i>	<i>TCGIResponse</i>
Windows CGI application	<i>TCGIApplication</i>	<i>TWinCGIRequest</i>	<i>TWinCGIResponse</i>

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 *TISAPIApplication*, which creates *TISAPIRequest* and *TISAPIResponse* 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 *TCGIApplication*, which creates *TCGIRequest* and *TCGIResponse* objects. Each request message is handled by a separate instance of the application.

Win-CGI stand-alone

A Win-CGI stand-alone Web server application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by *TCGIApplication*, which creates *TWinCGIRequest* and *TWinCGIResponse* objects. Each request message is handled by a separate instance of the application.

Creating Web server applications

All new Web server applications are created by selecting File | New from the menu of the main window and selecting Web Server Application in the New Items dialog. 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.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application and adds the required entries to the uses clause of the project file.

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 and containing an empty Web Module.

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 *TMIDASPageProducer*, 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 MIDAS application.

In addition to storing non-visual components and business rules, the Web module also acts as a 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

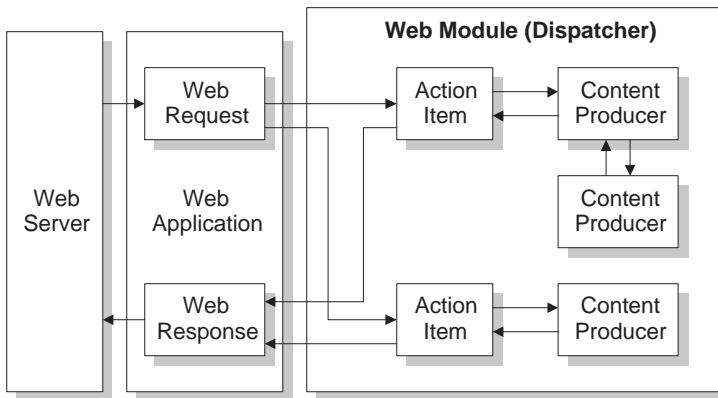
The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (either *TISAPIApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

Warning Do not include the forms unit in the project **uses** clause after the CGIApp or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

The structure of a Web server 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. It is described more fully in the section “The Web dispatcher” on page 29-8.

Figure 29.2 Structure of a Server Application

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. For more information on content producers, see “Generating the content of response messages” on page 29-17.

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. 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 chooses one or more action items to respond to the request.

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.

Action items are discussed in further detail in “Action items” on page 29-10.

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 looks through its list of action items for one that matches the *pathinfo* portion of the request message’s target URL and that can provide 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 properties 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” on page 14-30

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.

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 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:

Table 29.2 MethodType values

Value	Meaning
<i>mtGet</i>	The request is asking for the information associated with the target URI to be returned in a response message.
<i>mtHead</i>	The request is asking for the header properties of a response, as if servicing an <i>mtGet</i> request, but omitting the content of the response.
<i>mtPost</i>	The request is providing information to be posted to the Web application.
<i>mtPut</i>	The request asks that the resource associated with the target URI be replaced by the content of the request message.
<i>mtAny</i>	Matches any request method type, including <i>mtGet</i> , <i>mtHead</i> , <i>mtPut</i> , and <i>mtPost</i> .

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.

Caution 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.

Caution 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 re-evaluated 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 component 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" on page 29-17.

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 a *TWebRequest* object. In NSAPI and ISAPI applications, the request message is encapsulated by a *TISAPIRequest* object. Console CGI applications use *TCGIRequest* objects, and Windows CGI applications use *TWinCGIRequest* 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.

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.

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 also 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:

Value	What the message requests
<i>OPTIONS</i>	Information about available communication options.
<i>GET</i>	Information identified by the <i>URL</i> property.
<i>HEAD</i>	Header information from an equivalent GET message, without the content of the response.
<i>POST</i>	The server application to post the data included in the <i>Content</i> property, as appropriate.
<i>PUT</i>	The server application to replace the resource indicated by the <i>URL</i> property with the data included in the <i>Content</i> property.
<i>DELETE</i>	The server application to delete or hide the resource identified by the <i>URL</i> property.
<i>TRACE</i>	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 maintained 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* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In NSAPI and ISAPI applications, the response message is encapsulated by a *TISAPIResponse* object. Console CGI applications use *TCGIResponse* objects, and Windows CGI applications use *TWinCGIResponse* 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.

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.

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

Delphi 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 the following section.
- Table producers create HTML commands based on the information in a dataset. They are described in “Using database information in responses” on page 29-21.

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.

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
<i>Link</i>	<i>tgLink</i>	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an tag.
<i>Image</i>	<i>tgImage</i>	A graphic image. The result is an HTML tag.
<i>Table</i>	<i>tgTable</i>	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
<i>ImageMap</i>	<i>tgImageMap</i>	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.
<i>Object</i>	<i>tgObject</i>	An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag.
<i>Embed</i>	<i>tgEmbed</i>	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.

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:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
begin
    PageProducer1.HTMLFile := 'Greeting.html';
    Response.Content := PageProducer1.Content;
end;
```

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:

```
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;
```

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:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

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=1997>.
- Replaces <#TitleLine> with <#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=1997 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=1997> tag with the appropriate HTML 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.

As an alternate approach, special components on the Web MIDAS page of the component palette let you build Web servers that are part of a multi-tiered database application. See “Building Web applications using InternetExpress” on page 14-30 for details.

Adding a session to the Web module

Both console CGI applications and Win-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 *AutoSessionName* property 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 Chapter 16, "Managing database sessions."

Representing database information 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 tagname 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" on page 29-17.

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

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.

Setting up a dataset table producer

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.

Setting up a query table producer

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.

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. How you debug your Web server application depends on its type.

Debugging ISAPI and NSAPI applications

ISAPI and NSAPI 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. You will need to set your application's run parameters to launch the server. 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.

Debugging under Windows NT

Under Windows NT, you must have the correct user rights to debug a DLL. In the User Manager, add your name to the lists granting rights for

- Log on as Service
- Act as part of the operation system
- Generate security audits

Debugging with a Microsoft IIS server

To debug a Web server application using Microsoft IIS server (version 3 or earlier), choose Run | Parameters and set your application's run parameters as follows:

```
Host Application: c:\winnt\system32\inet_srv\inetinfo.exe
Run Parameters:  -e w3svc
```

This starts the IIS Server and allows you to debug your ISAPI DLL.

Note Your directory path for *inetinfo.exe* may differ from the example.

If you are using version 4 or later, you must first make some changes to the Registry and the IIS Administration service:

- 1 Use DCOMCnfg to change the identity of the IIS Admin Service to your user account.
- 2 Use the Registry Editor (REGEDIT) or other utility to remove the "LocalService" keyword from all IISADMIN-related subkeys under HKEY_CLASSES_ROOT/AppID and HKEY_CLASSES_ROOT/CLSID. This keyword appears in the following subkeys:

```
{61738644-F196-11D0-9953-00C04FD919C1} // IIS WAMREG admin Service
{9F0BD3A0-EC01-11D0-A6A0-00A0C922E752} // IIS Admin Crypto Extension
{A9E69610-B80D-11D0-B9B9-00A0C922E750} // IISADMIN Service
```

In addition, under the AppID node, remove the “RunAs” keyword from the first two of these subkeys. To the last subkey listed, add “Interactive User” as the value of the “RunAs” keyword.

- 3 Still using the Registry Editor, add “LocalService32” subkeys to all IISADMIN-related subkeys under the CLSID node. That is, for every subkey listed in step 2 (and any others under which you found the “LocalService” keyword), add a “LocalService32” subkey under the CLSID/<subkey> node. Set the default value of these new keys to

```
c:\winnt\system32\inetsrv\inetinfo.exe -e w3svc
```

(the path for inetinfo.exe may differ for your system).

- 4 Add a value of “dword:3” to the “Start” keyword for the following subkeys:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IISADMIN]
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MSDTC]
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC]
```

- 5 Stop the WWW, FTP, and IISAdmin services from the Microsoft Management Console or the Services dialog box in the control panel. As an alternative, you can simply do KILL INETINFO using KILL.EXE from the NT Resource Kit.

Now you are ready to debug in the same way as when using IIS version 3 or earlier. That is, choose Run | Parameters and set your application’s run parameters as follows:

```
Host Application: c:\winnt\system32\inetsrv\inetinfo.exe
Run Parameters: -e w3svc
```

Note When you have finished debugging, you will need to back out all the Registry changes you made in steps 2 through 4.

Debugging under MTS

Another approach you can take when using IIS is to configure your Web directory as an MTS Library package. You can then debug your ISAPI dll by running it under MTS.

To configure the Web directory as an MTS Library package, use the following steps:

- 1 Start the Internet Service Manager. You should see both the Internet Information Server and the Microsoft Transaction Server trees.
- 2 Expand the Internet Information Server tree to view the items under “Default Web Site”. Select the Web directory where your ISAPI dll is installed. Right click and choose Properties.
- 3 On the Virtual Directory tab page, check Run in separate memory space (isolated process), and click OK.
- 4 Expand the Microsoft Transaction Server tree to view the items under “Packages Installed”. Right click on the “Packages Installed” node and select Refresh.
- 5 You will see a package with the same suffix as the Web directory. Right click this package and choose Properties.

6 On the Identity tab page, select the Interactive User radio button, and click OK.

The previous steps configure your Web directory. After doing so, you can debug your ISAPI DLL as follows:

1 In Delphi, choose Run | Parameters. In the Host Application field, enter the fully qualified pathname of the MTS executable. Typically, this is

```
c:\winnt\system32\mtx.exe
```

2 In the parameters field, you must use the /p option with the name of the MTS package. To get this value, start the Internet Service Manager and expand the Microsoft Transaction Server tree to view the items under "Packages Installed". Right click on the package with the same suffix as the Web directory. Choose Properties, and copy the package name from the General tab page to the clipboard.

In the parameters field, paste the name of the package, and then surround it with double quotes and precede the quoted string with /p:. The resulting parameters field should look similar to the following:

```
/p:"IIS-{Default Web Site//ROOT/WEBPUB/DEMO}"
```

Note that there should not be a space between the colon and the package name.

Tip When the Web directory is installed as an MTS package, you can also use MTS to easily shut down the DLL. Just expand the Microsoft Transaction Server tree in the Internet Service Manager so that you can see the items under the "Packages Installed" node. Right click on the package with the same suffix as the Web directory and choose Shut Down.

Debugging with a Windows 95 Personal Web Server

To debug a Web server application using Personal Web Server, set your application's run parameters as follows:

```
Host Application: c:\Program Files\websvc\system\inetsw95.exe
Run Parameters: -w3svc
```

This starts the Personal Web Server and allows you to debug your ISAPI DLL.

Note Your directory path for *inetsw95.exe* may differ from the example.

Debugging with Netscape Server Version 2.0

Before using Web server applications on Netscape servers, you must make certain configuration changes.

First, copy the ISAPITER.DLL file (from the Bin directory) into the C:\Netscape\Server\Nsapi\Examples directory. (Your directory path may differ.)

Next, make the following modifications to the server configuration files located in the C:\Netscape\Server\Httpd-<servername>\Config directory.

1 In the OBJ.CONF file, insert the line

```
Init funcs="handle_isapi,check_isapi,log_isapi" fn="load_modules"
shlib="c:/netscape/server/nsapi/examples/ISAPIter.dll"
```

after the line

```
Init fn=load-types mime-types=mime.types
```

- 2 In the <Object name=default> section of OBJ.CONF, insert the lines

```
NameTrans from="/scripts" fn="pfx2dir" dir="C:/Netscape/Server/docs/scripts"
name="isapi"
```

before the line

```
NameTrans fn=document-root root="C:/Netscape/Server/docs"
```

- 3 Add the following section to the end of OBJ.CONF:

```
<Object name="isapi">
PathCheck fn="check_isapi"
ObjectType fn="force-type" type="magnus-internal/isapi"
Service fn="handle_isapi"
</Object>
```

- 4 Add the following line to the end of the MIME.TYPES file:

```
type=magnus-internal/isapi exts=dll
```

This should be the last line in the file.

Note Line breaks are included in steps 1 and 2 above only to enhance readability. Do not type carriage-returns when you place these lines in configuration files.

To debug a Web server application using Netscape Fast Track server, set the application's run parameters as follows:

```
Host Application: c:\Netscape\server\bin\httpd\httpd.exe
Run Parameters: c:\Netscape\server\httpd-<servername>\config
```

This starts the server and indicates to the server where the configuration files are located.

Debugging CGI and Win-CGI applications

It is more difficult to debug CGI and Win-CGI applications, because the application itself must be launched by the Web server.

Simulating the server

For Win-CGI applications, you can simulate the server by manually writing the configuration settings file that contains the request information. Then launch the Win-CGI application, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. You can then debug normally.

Debugging as a DLL

Another approach you can take with both CGI and Win-CGI applications is first to create and debug your application as an ISAPI or NSAPI application. Once your ISAPI or NSAPI application is working smoothly, convert it to a CGI or Win-CGI application. To convert your application, use the following steps:

- 1 Right-click the Web module and choose Add To Repository.
- 2 In the Add To Repository dialog, 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, choose CGI or Win-CGI, as appropriate.
- 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.

CGI and Win-CGI applications are simpler than ISAPI and NSAPI applications. Each instance of a CGI or Win-CGI application must handle only a single thread. Thus, these applications do not encounter the multi-threading issues that ISAPI and NSAPI applications must deal with. They also are immune to the problems that can arise from the caching of Web modules in ISAPI and NSAPI applications.

Working with sockets

This chapter describes the socket components that 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 actual networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as 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.

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.

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 applications that use it. You can copy the API of a standard third party server (such as ISAPI), or you can design and publish your own API.

Services and ports

Most standard services are associated, by convention, with specific port numbers. We will discuss port numbers in greater detail later. For now, consider the port number a numeric code for the service.

If you are implementing a standard service, Windows socket objects provide methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in a SERVICES file on Windows 95 or NT machines. See the Microsoft documentation for Windows sockets for more information on setting up a SERVICES file.

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 it wishes to connect to. 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 it is running on.
- 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 end points. 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.wSite.Com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. On Windows 95 and NT machines, if a host name is not available, you can create one for your local IP address by entering the name into the HOSTS file. See the Microsoft documentation on Windows sockets for more information on the HOSTS file.

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.

Earlier, we described port numbers as numeric codes for the services implemented by network applications. This is actually just 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 palette page includes two socket components (client sockets and server sockets) that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. Associated with each of these socket components are Windows socket objects, which represent the endpoint of an actual socket connection. The socket components use the Windows socket objects to encapsulate the Windows socket API 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 work with the Windows socket API calls, or customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the Windows socket objects.

Using client sockets

Add a client socket component (*TClientSocket*) to your form or data module to turn your application into a TCP/IP 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 Windows socket object (*TClientWinSocket*) to represent the client endpoint in a 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. You can specify the server system by its host name using the *Host* property. If you do not know the host name, or if you are concerned about the speed of locating the server, you can specify the IP address of the server system by using the *Address* property. You must specify either a host name or an IP address. If you specify both, the client socket component will use the host name.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can specify the server port number directly using the *Port* property, or indirectly by naming the desired service using the *Service* property. If you specify both the port number and the service, the client socket component will use the service name.

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 Windows socket object associated with your client socket component to obtain information about the connection. Use the *Socket* property to get access to the client Windows socket object. This Windows socket object has properties that enable you 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 *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. You can use the *Handle* property to access the window that receives messages from the socket connection. The *ASyncStyles* property determines what types of messages that window handle receives.

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 (*TServerSocket*) to your form or data module to turn your application into a TCP/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 Windows socket object (*TServerWinSocket*) to represent the server endpoint in a listening connection. It also uses a server client Windows socket object (*TServerClientWinSocket*) for the server endpoint of each active connection to a client socket that the server accepts.

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 *Port* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can specify the port number indirectly using the *Service* property. It is a good idea to use the *Service* property, as it is easy to miss typographical errors made when setting the port number. If you specify both the *Port* property and the *Service* property, the server socket will use the service name.

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 *OnClientConnect* event.

Getting information about connections

Once you have opened a listening connection with your server socket, you can use the server Windows socket object associated with your server socket component to obtain information about the connection. Use the *Socket* property to get access to the server Windows socket object. This Windows socket object has properties that enable you to find out about all the active connections to client sockets that were accepted by your server socket component. Use the *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. Use the *Handle* property to access the window that receives messages from the socket connection.

Each active connection to a client application is encapsulated by a server client Windows socket object (*TServerClientWinSocket*). You can access all of these through the *Connections* property of the server Windows socket object. These server client Windows socket objects have properties that enable you to determine the address and port number used by the client and server sockets which form the end points of the connection. You can use the *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. You can use the *Handle* property to access the window that receives messages from the socket connection. The *ASyncStyles* property determines what types of messages that window handle receives.

Closing server connections

When you want to shut down the listening connection, call the *Close* method. 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 clients shut down their individual connections to your server socket, you are informed by an *OnClientDisconnect* event.

Responding to socket events

When writing applications that use sockets, most of the work usually takes place in event handlers of the socket components. Some sockets generate events when it is time to begin reading or writing over the socket connection. These are described in “Reading and writing events” on page 30-10.

Client sockets receive an *OnDisconnect* event when the server ends a connection, and server sockets receive an *OnClientDisconnect* event when the client ends a connection.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive a number of events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, or if it should start reading or writing once the connection is formed, you will want to write event handlers to respond to these client events or server events.

Error events

Client sockets generate an *OnError* event when they receive error messages from the connection. Server sockets generate an *OnClientError*. You can write an *OnError* or *OnClientError* event handler to respond to these error messages. The event handler is passed information about

- What Windows 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:

- 1 An *OnLookup* event occurs prior to an attempt to locate the server socket. At this point you can not change the *Host*, *Address*, *Port*, or *Service* properties to change the server socket that is located. You can use the *Socket* property to access the client Windows socket object, and use its *SocketHandle* property to make Windows API calls that affect the client properties of the socket. For example, if you want to set the port number on the client application, you would do that now before the server client is contacted.
- 2 The Windows socket is set up and initialized for event notification.
- 3 An *OnConnecting* event occurs after the server socket is located. At this point, the Windows Socket object available through the *Socket* property can provide information about the server socket that will form the other end of the connection. This is the first chance to obtain the actual port and IP address used for the connection, which may differ from the port and IP address of the listening socket that accepted the connection.
- 4 The connection request is accepted by the server and completed by the client socket.
- 5 An *OnConnect* event occurs after the connection is established. If your socket should immediately start reading or writing over the connection, write an *OnConnect* event handler to do it.

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 *OnListen* event occurs. At this point you can obtain the server Windows socket object through the *Socket* property. You can use its *SocketHandle* property to make changes to the socket before it is opened for listening. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListen* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- 1 The server socket generates an *OnGetSocket* event, passing in the Windows socket handle for the socket that forms the server endpoint of the connection. If you want to provide your own customized descendant of *TServerClientWinSocket*, you can create one in an *OnGetSocket* event handler, and that will be used instead of *TServerClientWinSocket*.
- 2 An *OnAccept* event occurs, passing in the new *TServerClientWinSocket* object to the event handler. This is the first point when you can use the properties of *TServerClientWinSocket* to obtain information about the server endpoint of the connection to a client.

- 3 If *ServerType* is *stThreadBlocking* an *OnGetThread* event occurs. If you want to provide your own customized descendant of *TServerClientThread*, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerClientThread*. For more information on creating custom server client threads, see “Writing server threads” on page 30-13.
- 4 If *ServerType* is *stThreadBlocking*, an *OnThreadStart* event occurs as the thread begins execution. If you want to perform any initialization of the thread, or make any Windows socket API calls before the thread starts reading or writing over the connection, use the *OnThreadStart* event handler.
- 5 The client completes the connection and an *OnClientConnect* 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 your network application. To create a non-blocking connection

- On client sockets, set the *ClientType* property to *ctNonBlocking*.
- On server sockets, set the *ServerType* property to *stNonBlocking*.

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 that inform your socket when it needs to read or write over the connection. With client sockets, you can respond to these notifications in an *OnRead* or *OnWrite* event handler. With server sockets, you can respond to these events in an *OnClientRead* or *OnClientWrite* event handler.

The Windows socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This Windows 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 *ReceiveText* method. Before using the *ReceiveBuf* method, use the *ReceiveLength* method to get an estimate of the number of bytes the socket on the other end of the connection is ready to send.

To write to the socket connection, use the *SendBuf*, *SendStream*, or *SendText* method. If you have no more need of the socket connection after you have written your information over the socket, you can use the *SendStreamThenDrop* method. *SendStreamThenDrop* closes the socket connection after writing all information that can be read from the stream. If you use the *SendStream* or *SendStreamThenDrop* method, do not free the stream object. The socket frees the stream automatically when the connection is closed.

Note *SendStreamThenDrop* will close down a server connection to an individual client, not a listening connection.

Blocking connections

When the connection is blocking your socket must initiate reading or writing over the connection rather than waiting 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 sockets, set the *ClientType* property to *ctBlocking* 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 *ServerType* property to *stThreadBlocking* 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 *ServerType* is *stThreadBlocking*.

Using threads with blocking connections

Client sockets do not automatically spawn new threads when reading or writing using a blocking connection. If your client application has nothing else to do until the information has been read or written, this is what you want. If your application includes a user interface that must still respond to the user, however, you will want to generate a separate thread for the reading or writing.

When server sockets form blocking connections, they always spawn separate threads for every client connection, so that no client must wait until another client has finished reading or writing over the connection. By default, server sockets use *TServerClientThread* objects to implement the execution thread for each connection.

TServerClientThread objects simulate the *OnClientRead* and *OnClientWrite* events that occur with non-blocking connections. However, these events occur on the listening socket, which is not thread-local. If client requests are frequent, you will want to create your own descendent of *TServerClientThread* to provide thread-safe reading and writing.

Using *TWinSocketStream*

When implementing the thread for a blocking connection, you must determine when the socket on the other end of the connection is ready for reading or writing. Blocking connections do not notify the socket when it is time to read or write. To see if the connection is ready, use a *TWinSocketStream* object. *TWinSocketStream* provides methods to help coordinate the timing of reading and writing. Call the *WaitForData* method to wait until the socket on the other end is ready to write.

When reading or writing using *TWinSocketStream*, the stream times out if the reading or writing has not completed after a specified period of time. As a result of this timing out, the socket application won't hang endlessly trying to read or write over a dropped connection.

Note You can not use *TWinSocketStream* with a non-blocking connection.

Writing client threads

To write a thread for client connections, define a new thread object using the New Thread Object dialog. For more information, see “Defining thread objects” on page 8-1.

The *Execute* method of your new thread object handles the details of reading and writing over the thread connection. It creates a *TWinSocketStream* object, and uses that to read or write. For example:

```

procedure TMyClientThread.Execute;
var
    TheStream: TWinSocketStream;
    buffer: string;
begin
    { create a TWinSocketStream for reading and writing }
    TheStream := TWinSocketStream.Create(ClientSocket1.Socket, 60000);
    try
        { fetch and process commands until the connection or thread is terminated }
        while (not Terminated) and (ClientSocket1.Active) do
            begin
                try
                    GetNextRequest(buffer); { GetNextRequest must be a thread-safe method }
                    { write the request to the server }
                    TheStream.Write(buffer, Length(buffer) + 1);
                    { continue the communication (eg read a response from the server) }
                    ...
                except
                    if not (ExceptObject is EAbort) then
                        Synchronize(HandleThreadException); { you must write HandleThreadException }
                end;
            end;
    end;

```

```

    finally
        TheStream.free;
    end;
end;

```

To use your thread, create it in an *OnConnect* event handler. For more information about creating and running threads, see “Executing thread objects” on page 8-10.

Writing server threads

Threads for server connections are descendents of *TServerClientThread*. Because of this, you can’t use the New Thread object dialog. Instead, declare your thread manually as follows:

```
TMyServerThread = class(TServerClientThread);
```

To implement this thread, you override the *ClientExecute* method instead of the *Execute* method.

Implementing the *ClientExecute* method is much the same as writing the *Execute* method of the thread for a client connection. However, instead of using a client socket component that you place in your application from the Component palette, the server client thread must use the *TServerClientWinSocket* object that is created when the listening server socket accepts a client connection. This is available as the public *ClientSocket* property. In addition, you can use the protected *HandleException* method rather than writing your own thread-safe exception handling. For example:

```

procedure TMyServerThread.ClientExecute;
var
    Stream : TWinSocketStream;
    Buffer : array[0 .. 9] of Char;
begin
    { make sure connection is active }
    while (not Terminated) and ClientSocket.Connected do
    begin
        try
            Stream := TWinSocketStream.Create(ClientSocket, 60000);
            try
                FillChar(Buffer, 10, 0); { initialize the buffer }
                { give the client 60 seconds to start writing }
                if Stream.WaitForData(60000) then
                begin
                    if Stream.Read(Buffer, 10) = 0 then { if can't read in 60 seconds }
                        ClientSocket.Close;           { close the connection }
                    { now process the request }
                    ...
                end
                else
                    ClientSocket.Close; { if client doesn't start, close }
                finally
                    Stream.Free;
                end;
            except
                HandleException;
            end;
        end;
    end;

```

Warning Server sockets cache the threads they use. Be sure the *ClientExecute* method performs any necessary initialization so that there are no adverse results from changes made when the thread last executed.

To use your thread, create it in an *OnGetThread* event handler. When creating the thread, set the *CreateSuspended* parameter to *False*.

Creating custom components

The chapters in “Creating custom components” present concepts necessary for designing and implementing custom components in Delphi.

Overview of component creation

This chapter 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.

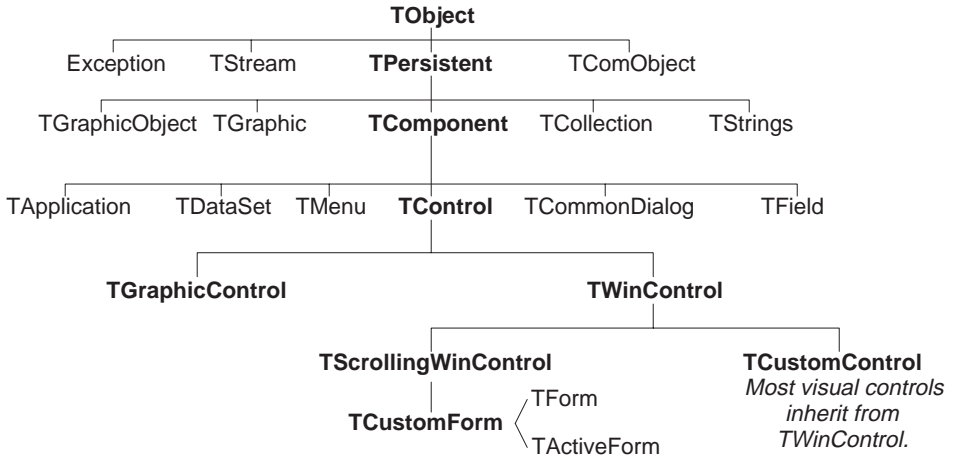
- Visual Component Library
- Components and classes
- How do you create components?
- What goes into a component?
- Creating a new component
- Testing uninstalled components

For information on installing new components, see “Installing component packages” on page 9-6.

Visual Component Library

Delphi’s components are all part of a class hierarchy called the Visual Component Library (VCL). Figure 31.1 shows the relationship of selected classes that make up the VCL. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Chapter 32, “Object-oriented programming for component writers.”

The *TComponent* class is the shared ancestor of every component in the VCL. *TComponent* provides the minimal properties and events necessary for a component to work in Delphi. The various branches of the library provide other, more specialized capabilities.

Figure 31.1 Visual Component Library class hierarchy

When you create a component, you add to the VCL 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.

How do you create 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 from any existing component, but the following are the most common ways to create components:

- Modifying existing controls
- Creating windowed controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

Table 31.1 summarizes the different kinds of components and the classes you use as starting points for each.

Table 31.1 Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as <i>TButton</i> or <i>TListBox</i> , or an abstract component type, such as <i>TCustomListBox</i>
Create a windowed control	<i>TWinControl</i>
Create a graphic control	<i>TGraphicControl</i>
Subclassing a Windows control	Any Windows control
Create a nonvisual component	<i>TComponent</i>

You can also derive classes that are not components and cannot be manipulated on a form. Delphi includes many such classes, like *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 with Delphi.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the VCL 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 VCL 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.

Chapter 33, “Creating properties,” explains publishing inherited properties. Chapter 39, “Modifying an existing component,” and Chapter 41, “Customizing a grid,” show examples of modifying existing controls.

Creating windowed controls

Windowed controls 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 Windows identify and operate on the control. The handle allows the control to receive input focus and can be passed to Windows API functions.

All windowed controls descend from the *TWinControl* class. These include most standard Windows 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.

Chapter 41, “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.

Delphi supports the creation of 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 handles *WM_PAINT* messages; all you need to do is override the *Paint* method.

Chapter 40, “Creating a graphic component,” 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.

Using Delphi, 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*) 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
- Properties, methods, and events
- Graphics encapsulation
- Registration

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 excellent 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 window or 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, there is no 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.

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 chapter devoted to it in this book, 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.

Chapter 33, “Creating properties,” explains how to add properties 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.

Chapter 34, “Creating events,” explains how to implement standard events and how to define new ones.

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.

Chapter 35, "Creating methods," explains how to add methods to your components.

Graphics encapsulation

Delphi simplifies Windows graphics by encapsulating various graphic 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 much 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 also 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. Graphics features are detailed in Chapter 36, "Using graphics in components."

Registration

Before you can install your components in the Delphi IDE, you have to register them. Registration tells Delphi where you want your component to appear on the Component palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Chapter 38, "Making components available at design time."

Creating a new component

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the Component 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 Component 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 Creating a unit for the new component.
- 2 Deriving your component from an existing component type.
- 3 Adding properties, methods, and events.
- 4 Registering your component with Delphi.
- 5 Creating a Help file for your component and its properties, methods, and events.
- 6 Creating a package (a special dynamic-link library) so that you can install your component in the Delphi IDE.

When you finish, the complete component includes these 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

Creating a help file to instruct component users on how to use the component is optional. Including a help file is mandatory only if one is created.

The chapters in the rest of Part IV explain all the aspects of building components and provide several complete examples of writing different kinds of components.

Using the Component wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify only these things:

- The class from which it is derived
- The class name for the new component
- The Component palette page you want it to appear on
- 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:

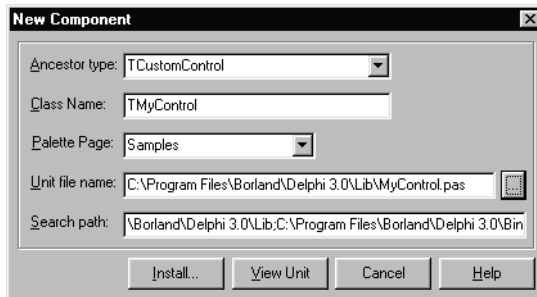
- Creating a unit
- Deriving the component
- Registering the component

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To start the Component wizard, choose one of these two methods:

- Choose Component | New Component.
- Choose File | New and double-click on Component

Figure 31.2 Component wizard



Fill in the fields in the Component wizard:

- 1 In the Ancestor Type field, specify the class from which you are deriving your new component.
- 2 In the Class Name field, specify the name of your new component class.
- 3 In the Palette Page field, specify the page on the Component palette on which you want the new component to be installed.
- 4 In the Unit file name field, specify the name of the unit you want the component class declared in.
- 5 If the unit is not on the search path, edit the search path in the Search Path field as necessary.

To place the component in a new or existing package, click Component | Install and use the dialog box that appears to specify a package.

Warning If you derive a component from a VCL 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 Unit. (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:

```
unit MyControl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

```

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]);
end;

end.

```

Creating a component manually

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

- 1 Creating a unit file
- 2 Deriving the component
- 3 Registering the component

Creating a unit file

A unit is a separately compiled module of Object Pascal 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 unit, choose File | New to and double-click on Unit. Delphi creates a new unit file and opens it in the Code editor.

To open an existing unit, choose File | Open and select the source code unit that you want to add your component to.

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 Component palette.

Once you have either a new or existing unit for your component, you can 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. “How do you create components?” on page 31-2 describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in the section “Defining new classes” on page 32-1.

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*.

To create a simple component class, add the following class declaration to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.

Registering the component

Registration is a simple process that tells Delphi which components to add to its component library, and on which pages of the Component palette they should appear. For a more detailed discussion of the registration process, see Chapter 38, “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:

```
procedure Register;
```

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.

- 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 Component palette page 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*.

To register a component named *TMyControl* and place it on the Samples page of the palette, you would add the following *Register* procedure to the unit that contains *TMyControl*’s declaration:

```

procedure Register;
begin
    RegisterComponents('Samples', [TNewControl]);
end;

```

This *Register* procedure places *TMyControl* on the Samples page of the Component palette.

Once you register a component, you can compile it into a package (see Chapter 20) and install it on the Component palette.

Testing uninstalled components

You can test the runtime behavior of a component before you install it on the Component 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 Component palette.

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.

Suppose you want to test a new component of type *TMyControl* in a unit named *MyControl*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```

unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;           { 1. Add NewTest to uses clause }
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);           { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1: TMyControl1;                       { 2. Add an object field }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);           { 4. Construct the component }
  MyControl1.Parent := Self;                       { 5. Set Parent property if component is a control }
  MyControl1.Left := 12;                           { 6. Set other properties... }
  :
  :
  ...continue as needed }
end;
end.
```


Object-oriented programming for component writers

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 this chapter, 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.

To change 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 Component 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.

Chapter 39, “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*.

To add 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.

Chapter 41, “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. Table 31.1 on page 31-3 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.

Here is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

A finished component declaration usually includes property, event, and method declarations before the **end**. But a declaration like the one above is also valid, and provides a starting point for the addition of component features.

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 32.1, “Levels of visibility within an object,” shows the levels of visibility, from most restrictive to most accessible:

Table 32.1 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 from the Object Inspector.	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.

Here is an example that shows how declaring a field as **private** hides it from application developers. The listing shows two form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```
unit HideInfo;
interface
```

```

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;

type
  TSecretForm = class(TForm)                                { declare new form }
    procedure FormCreate(Sender: TObject);
  private                                                  { declare private part }
    FSecretCode: Integer;                                   { declare a private field }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                       { this compiles correctly }
end;
end.                                                       { end of unit }

unit TestHide;                                           { this is the main form file }

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo;                                               { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;                            { compiler stops with "Field identifier expected" }
end;
end.                                                       { end of unit }

```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

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.

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.

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

```

type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;           { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;           { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
:
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;

```

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.

Here is an example of a published property called *Temperature*. Because it is published, it appears in the Object Inspector at design time.

```

type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;           { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;   { writable! }
  end;

```

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

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.

An example of static methods

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;      { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;      { this is also different }
  end;
```

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).

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```

type
  TFirstComponent = class(TCustomControl)
    procedure Move;           { static method }
    procedure Flash; virtual; { virtual method }
    procedure Beep; dynamic;  { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;           { declares new method }
    procedure Flash; override; { overrides inherited method }
    procedure Beep; override; { overrides inherited method }
  end;

```

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 must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. Delphi cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Chapter 33, "Creating properties," and Chapter 35, "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

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, you let Delphi read the values from the Object Inspector. 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.

Table 33.1 How properties appear in the Object Inspector

Property type	Object Inspector 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 <i>TPersistent</i> .
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. Redefining means adding a declaration for the inherited property to the declaration of the descendant class.

If you derive a component from *TWinControl*, for example, it inherits the protected *Ctl3D* property. By redeclaring *Ctl3D* in your new component, you can change the level of protection to either public or published.

The following code shows a redeclaration of *Ctl3D* as published, making it available at design time.

```
type
  TSampleComponent = class(TWinControl)
    published
      property Ctl3D;
    end;
```

When you redeclare a property, you specify only the property name, not the type and other information described below in “Defining properties”. You can also declare new default values and specify whether to store the property.

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

Defining properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

The property declaration

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.

Here is a typical declaration for a property called *Count*.

```
type
  TYourComponent = class(TComponent)
```

```

private
    FCount: Integer;           { used for internal storage }
    procedure SetCount (Value: Integer); { write method }
public
    property Count: Integer read FCount write SetCount;
end;

```

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.

The following component-type declaration shows a property that uses direct access for both the **read** and the **write** parts.

```

type
    TSampleComponent = class(TComponent)
    private
        FMyProperty: Boolean;           { internal storage is private }
        { declare field to hold property value }
    published
        { make property available at design time }
        property MyProperty: Boolean read FMyProperty write FMyProperty;
    end;

```

Access methods

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.

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

```
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;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ;
```

Because each element of the date (day, month, and year) is an int, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

```
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;
```

This is the write method that sets the appropriate date element:

```
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;

```

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” on page 33-8), 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 *Object Pascal 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 *Object Pascal Language Guide*.)

If you do not declare a write method, the property is read-only. For a published property to be usable at design time, it must be defined as read/write.

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*.

```

procedure TMyComponent.SetCount(Value: Integer);
begin
    if Value <> FCount then

```

```

begin
    FCount := Value;
    Update;
end;
end;

```

Default property values

When you declare a property, you can specify a *default value* for it. Delphi 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, Delphi 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,

```
property Cool Boolean read GetCool write SetCool 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,

```
property FavoriteFlavor string 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.

Here is the declaration of a component that includes a single Boolean property called *IsTrue* with a default value of *True*. Below the declaration (in the **implementation** section of the unit) is the constructor that initializes the property.

```

type
    TSampleComponent = class(TComponent)
    private
        FIsTrue: Boolean;
    public
        constructor Create(AOwner: TComponent); override;
    published
        property IsTrue: Boolean read FIsTrue write FIsTrue default True;
    end;
:
constructor TSampleComponent.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);           { call the inherited constructor }
    FIsTrue := True;                    { set the default value }
end;

```

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.

The following example shows the declaration of a property that returns a string based on an integer index.

```

type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
:
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
end;

```

Storing and loading properties

Delphi stores forms and their components in form (.DFM) files. A form file is a binary representation of 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 public and 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 redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

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.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```

type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;           { override to set new default }
  published
    property Align default alBottom;                             { redeclare with new default value }
  end;
:
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                     { perform inherited initialization }
  Align := alBottom;                                           { assign new default value for Align }
end;

```

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 at runtime 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 method.

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean method:

```

type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
    property Important: Integer stored True;                     { normally not stored }
    property Unimportant: Integer stored False;                 { always stored }
  published
    property Sometimes: Integer stored StoreIt;                 { normally stored always }
    property Unimportant: Integer stored False;                 { never stored }
    property Sometimes: Integer stored StoreIt;                 { storage depends on function value }
  end;

```

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*:

```

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;

```

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:

```

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
                TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
                Result := True
            else Result := False;
        end;
    end;

```

```
    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;
```


Creating events

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

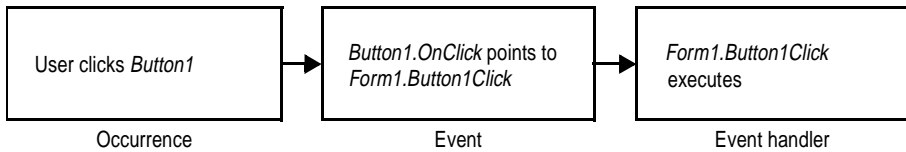
Events are implemented as properties, so you should already be familiar with the material in Chapter 33, “Creating properties,” before you attempt to create or change a component’s events.

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, Delphi 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 method pointers

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in an instance object. 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 an object. 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 in a specific instance object. That object is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

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:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;           { declare a field to hold the method pointer }
    :
  protected
```



```

    property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;

```

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 an object.

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 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” on page 34-8.)

Events happen almost constantly in a Windows application. Just moving the mouse pointer across a visual component makes Windows send 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 VCL 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 Delphi inherit events for the most common Windows 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 table lists events available in all controls:

<i>OnClick</i>	<i>OnDragDrop</i>	<i>OnEndDrag</i>	<i>OnMouseMove</i>
<i>OnDbClick</i>	<i>OnDragOver</i>	<i>OnMouseDown</i>	<i>OnMouseUp</i>

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:

<i>OnEnter</i>	<i>OnKeyDown</i>	<i>OnKeyPress</i>
<i>OnKeyUp</i>	<i>OnExit</i>	

Like the standard events in *TControl*, the windowed-control events have corresponding methods.

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.

For example, to create a component that surfaces the *OnClick* event at design time, you would add the following to the component's class declaration.

```
type
  TMyControl = class(TCustomControl)
  :
  published
    property OnClick;
  end;
```

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.

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as an application developer would, you override the protected method *Click*:

```

procedure click override           { forward declaration }
:
:
procedure TMyControl.Click;
begin
  inherited Click;                 { perform standard handling, including calling handler }
  ...                               { your customizations go here }
end;

```

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.

But 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.

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.

An empty handler should produce the same result as no handler at all. So the code for calling an application's event handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
... { perform default handling }
```

You should *never* have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

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.

When handling key-press events, for example, application developers can suppress the component's default handling of the keystroke by setting the `var` parameter *Key* to a null character (#0). The logic for supporting this looks like

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ... { perform default handling }
```

The actual code is a little different from this because it deals with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Creating methods

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 environment and are accessible at design time.

Avoiding dependencies

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 might 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” on page 32-4. 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 Windows 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 VCL, 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 descendent 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 descendent components; it forces Component writers to redeclare the abstract member in descendent 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, you do two things:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

The following code shows a component that defines two new methods, one protected static method and one public virtual method.

```

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

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 the VCL font cache.

Here are two examples of how simple Delphi's graphics code can be. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window in an application written with ObjectWindows. The second uses a canvas to draw the same ellipse in an application written with Delphi.

```

procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
    PenHandle, OldPenHandle: HPEN;
    BrushHandle, OldBrushHandle: HBRUSH;
begin
    PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));           { create blue pen }
    OldPenHandle := SelectObject(PaintDC, PenHandle);           { tell DC to use blue pen }
    BrushHandle := CreateSolidBrush(RGB(255, 255, 0));           { create a yellow brush }
    OldBrushHandle := SelectObject(PaintDC, BrushHandle);       { tell DC to use yellow brush }
    Ellipse(HDC, 10, 10, 50, 50);                               { draw the ellipse }
    SelectObject(OldBrushHandle);                                { restore original brush }
    DeleteObject(BrushHandle);                                   { delete yellow brush }
    SelectObject(OldPenHandle);                                  { restore original pen }
    DeleteObject(PenHandle);                                     { destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
        begin
            Pen.Color := clBlue;                                 { make the pen blue }
            Brush.Color := clYellow;                             { make the brush yellow }
            Ellipse(10, 10, 50, 50);                             { draw the ellipse }
        end;
end;

```

Using the canvas

The canvas class encapsulates Windows graphics 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 low-level access to the Windows GDI.

Table 36.1 summarizes the capabilities of the canvas.

Table 36.1 Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as <i>MoveTo</i> , <i>LineTo</i> , <i>Rectangle</i> , and <i>Ellipse</i>
	Displaying and measuring text	<i>TextOut</i> , <i>TextHeight</i> , <i>TextWidth</i> , and <i>TextRect</i> methods
	Filling areas	<i>FillRect</i> and <i>FloodFill</i> methods
Intermediate	Customizing text and graphics	<i>Pen</i> , <i>Brush</i> , and <i>Font</i> properties
	Manipulating pixels	<i>Pixels</i> property.
	Copying and merging images	<i>Draw</i> , <i>StretchDraw</i> , <i>BrushCopy</i> , and <i>CopyRect</i> methods; <i>CopyMode</i> property
Low	Calling Windows GDI functions	<i>Handle</i> property

For detailed information on canvas classes and their methods and properties, see online Help.

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.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

The picture recognizes *.BMP* as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Handling palettes

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 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 Windows 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.

Table 36.2 summarizes the image-copying methods in canvas objects.

Table 36.2 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

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.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

```

type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
  ..
implementation
  ..
constructor TShape.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 TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                       { erase and repaint the component }
end;

```


Handling messages

One of the keys to traditional Windows programming is handling the *messages* sent by Windows to applications. Delphi handles most of the common ones for you. It is possible, however, that you will need to handle messages that Delphi does not already handle or that you will create your own messages.

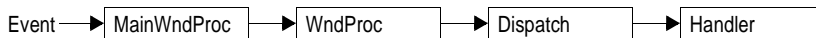
There are three aspects to working with messages:

- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

Understanding the message-handling system

All Delphi 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” on page 37-6 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 API what each parameter contained. More recently, 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*.

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.

Delphi 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

Delphi 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. Delphi 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, 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.

For example, to override a component's handling of the `WM_PAINT` message, you redeclare the `WMPaint` method:

```
type
  TMyComponent = class(...)
  :
  procedure WMPaint (var Message: TWMPaint); message WM_PAINT;
end;
```

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`.

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:

```
procedure TMyControl.WndProc (var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc (Message);
end;
```

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.

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 Delphi provides handlers for most common Windows 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 two aspects:

- Defining your own messages
- Declaring a new message-handling method

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 Windows messages and notification of state changes.

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.

The following code shows two user-defined messages.

```

const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;

```

Declaring a message-record 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*.

For example, here is the message record for all mouse messages, *TWMMouse*, which uses a variant record to define two sets of names for the same parameters.

```

type
  TWMMouse = record
    Msg: TMsgParam;      ( first is the message ID )
    Keys: Word;          ( this is the wParam )
    case Integer of
      0: {
          XPos: Integer;  ( either as x- and y-coordinates... )
          YPos: Integer;
        }
      1: {
          Pos: TPoint;    ( ... or as a single point )
          Result: Longint; ( and finally, the result field )
        }
    end;

```

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.

Here is the declaration, for example, of a message handler for a user-defined message called *CM_CHANGECOLOR*.

```
const
  CM_CHANGECOLOR = WM_APP + 400;

type
  TMyComponent = class(TControl)
    :
  protected
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
  end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
```


Making components available at design time

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- 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" on page 9-6.

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 Component 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.

The following code shows the outline of a simple unit that creates and registers new components:

```
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.
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component 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 Component 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 page of the Component 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.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
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;
```

Specifying the palette page

The palette-page name is a string. If the name you give for the palette page does not already exist, Delphi creates a new page with that name. Delphi stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

Using the RegisterComponents function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes three parameters: the name of a Component palette page, the array of component classes, and the index of the last entry in the array.

Set the Page parameter to the name of the page on the component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, Delphi creates a new palette page 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 component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);          {add to system page}
  RegisterComponents('MyCustomPage', [TCustom1, TCustom2]);    { new page }
end;
```

Adding palette bitmaps

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, Delphi uses a default bitmap.

Because the palette bitmaps are needed only at design time, you don't compile them into the component's compilation unit. Instead, you supply them in a Windows resource file with the same name as the unit, but with the extension *.DCR* (dynamic

component resource). You can create this resource file using the Image editor in Delphi. Each bitmap should be 24 pixels square.

For each component you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled files, so Delphi can find the bitmaps when it installs the components on the Component palette.

For example, if you create a component named *TMyControl* in a unit named *ToolBox*, you need to create a resource file called *TOOLBOX.DCR* that contains a bitmap called *TMYCONTROL*. The resource names are not case-sensitive, but by convention they are usually in uppercase letters.

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" on page 38-4 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 Adding component help files

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:

1 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.

2 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.

3 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 component help files

To add your Help file to Delphi, use the OpenHelp utility (called oh.exe) located in the bin directory or accessed using Help | Customize in the IDE.

You will find information about using OpenHelp in the OpenHelp.hlp file, including adding your Help file to the Help system.

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

The *DsgnIntf* unit defines 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 Table 38.1.

The *DsgnIntf* 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.

Table 38.1 Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from <i>TOrdinalProperty</i> .
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	<i>Char</i> -type and subranges of <i>Char</i> , 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.

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

Editing the property as text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor classes provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown in Table 38.2.

Table 38.2 Methods for reading and writing property values

Property type	Get method	Set method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

When you override a *GetValue* method, you will call one of the Get methods, and when you override *SetValue*, you will call one of the Set methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns *unknown*.

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property is not a string value, *GetValue* must convert the value into a string representation.

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.

Here are the *GetValue* and *SetValue* methods for *TIntegerProperty*. *Integer* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
    Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
var
    L: Longint;
begin
    L := StrToInt(Value);                { convert string to number }
    with GetTypeData(GetPropType)^ do   { this uses compiler data for type Integer }
        if (L < MinValue) or (L > MaxValue) then { make sure it's in range... }
            raise EPropertyError.Create(        { ...otherwise, raise exception }
                FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));
        SetOrdValue(L);                    { if in range, go ahead and set value }
    end;
end;
```

The specifics of the particular examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates 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.

The *Color* properties found in most components use the standard Windows color dialog box as a property editor. Here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```

procedure TColorProperty.Edit;
var
    ColorDialog: TColorDialog;
begin
    ColorDialog := TColorDialog.Create(Application);           { construct the editor }
    try
        ColorDialog.Color := GetOrdValue;                    { use the existing value }
        if ColorDialog.Execute then                        { if the user OKs the dialog... }
            SetOrdValue(ColorDialog.Color);                  { ...use the result to set value }
    finally
        ColorDialog.Free;                                   { destroy the editor }
    end;
end;

```

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:

Table 38.3 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.

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList];
end;
```

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)*.
- 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.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you have created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are **nil** and an empty string, respectively.

- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property (of type *TComponentName*) of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

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.

```
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;
```

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.

```
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;
```

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:

```
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;
```

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:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

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.

```

procedure TMyComponent.Copy;
var
    MyFormat : Word;
    AData,APalette : THandle;
begin
    TImage(Component).Picture.Bitmap.SaveToClipboardFormat(MyFormat, AData, APalette);
    Clipboard.SetAsHandle(MyFormat, AData);
end;

```

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*:

```
RegisterComponentEditor(TMyComponent, 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.

```

procedure Register;
begin
    RegisterComponents('Miscellaneous', [TMyComponent]);
    RegisterComponentEditor(classes[0], TMyEditor);
end;

```

Property categories

In the Delphi IDE, the Object Inspector affords the programmer the ability to selectively hide and display properties based on property categories. The properties of new custom components can also be fitted into this scheme by registering properties in categories. Do this at the same time the component is being registered by calling one of the property registration functions *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use the former to register a single property. Use the latter to register multiple properties in a single function call. These functions are defined in the unit *DsgnIntf*.

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 simply deemed to be in the *TMiscellaneousCategory* category. These properties would be displayed or hidden in the Object Inspector based on that default categorization.

Delphi supplies thirteen stock property categories, in the form of property classes. Register a property of a new custom component in one of these provided categories or create your own property category classes based on these built-in classes.

In addition to these two functions for registering properties, there is an `IsPropertyInCategory` function. This function is useful for such endeavors as creating localization utilities, in which you must determine whether a property is registered in a given property category.

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 allows you to 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".

```
RegisterPropertyInCategory(TVisualCategory, 'AutoSize');
```

The second variation identifies the property using the characteristics component class type and property name. The example below registers (into the category `THelpCategory`) a property named "HelpContext" of a component of the custom class `TMyButton`.

```
RegisterPropertyInCategory(THelpCategory, TMyButton, 'HelpContext');
```

The third variation uses the property's type and the property's name to identify the property. The example below registers a property based on a combination of its type, `Integer`, and its name, "Width".

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer), 'Width');
```

The last variation identifies the property using only its property type. The example below registers a property based on its type, `Integer`.

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer));
```

See the section [Property category classes](#) 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 allows you to identify properties for association with a property category based on property name. A list of property names is passed as an array of `String`. Each property identified by name in the list is registered with the specified

property category. In the example below, four properties are registered in the category `THelpCategory`. These four properties are identified by name using the strings “HelpContext”, “Hint”, “ParentShowHint”, and “ShowHint”.

```
RegisterPropertiesInCategory(THelpCategory, ['HelpContext', 'Hint', 'ParentShowHint',
    'ShowHint']);
```

The second variation identifies the properties by their type. In the example below, all of the properties in the custom component that are of type `String` are registered in the category `TLocalizableCategory`.

```
RegisterPropertiesInCategory(TLocalizableCategory, TypeInfo(String));
```

The third variation allows you to pass a list of various criteria, not all of which need be the same type, to use to identify properties to register. 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 `TLocalizableCategory`.

```
RegisterPropertiesInCategory(TLocalizableCategory, ['Text', TEdit]);
```

See the section `Property category classes` for a list of the available property categories and a brief description of their uses.

Property category classes

Built-in property categories

Delphi provides a built-in set of twelve property categories with which you can associate properties in custom components. Use one of these property category class names for the `ACategoryClass` parameter of the `RegisterPropertyInCategory` and `RegisterPropertiesInCategory` functions.:

Table 38.4 Property categories

Category	Purpose
<i>TActionCategory</i>	Properties related to runtime actions; the Enabled and Hint properties of <code>TEdit</code> are in this category.
<i>TDatabaseCategory</i>	Properties related to database operations; the <code>DatabaseName</code> and <code>SQL</code> properties of <code>TQuery</code> are in this category.
<i>TDragNDropCategory</i>	Properties related to drag-n-drop and docking operations; the <code>DragCursor</code> and <code>DragKind</code> properties of <code>TImage</code> are in this category.
<i>THelpCategory</i>	Properties related to using online help or hints; the <code>HelpContext</code> and <code>Hint</code> properties of <code>TMemo</code> are in this category.
<i>TLayoutCategory</i>	Properties related to the visual display of a control at design-time; the <code>Top</code> and <code>Left</code> properties of <code>TDBEdit</code> are in this category.
<i>TLegacyCategory</i>	Properties related to obsolete operations; the <code>Ctl3D</code> and <code>ParentCtl3D</code> properties of <code>TComboBox</code> are in this category.
<i>TLinkageCategory</i>	Properties related to associating or linking one component to another; the <code>DataSet</code> property of <code>TDataSource</code> is in this category.
<i>TLocaleCategory</i>	Properties related to international locales; the <code>BiDiMode</code> and <code>ParentBiDiMode</code> properties of <code>TMainMenu</code> are in this category.

Table 38.4 Property categories (continued)

Category	Purpose
<i>TLocalizableCategory</i>	Properties related to database operations; the DatabaseName and SQL properties of TQuery are in this category.
<i>TMiscellaneousCategory</i>	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.
<i>TVisualCategory</i>	Properties related to the visual display of a control at runtime; the Align and Visible properties of TScrollBar are in this category.
<i>TInputCategory</i>	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.

Deriving new property categories

You can create new property categories of your own design by deriving a class from either the base class *TPropertyCategory* or one of the built-in descendants. See the section Property category classes for a list of the available property categories and a brief description of their uses.

When deriving a new property category class, override the Name method. The Name method provides the name of the category for display in the Object Inspector. This method must be superseded with a method that returns the name of the custom category. The Name method may simply return a String value or it may retrieve a value from a resource. The latter is useful for easily internationalizing a custom component and its categories.

Given the new Name method below for a custom category class, the text “My Special” would appear in the Object Inspector when property categories are displayed (and at least one of the current object’s properties is registered in this property class).

```
class function TMySpecialCategory.Name: String;
begin
    Result := 'My Special';
end;
```

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 allows you to 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 the class TEdit, have the name “Text”, and be in the property category TLocalizableCategory.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, TEdit, 'Text');
```

The second variation allows you to 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 belong to the class `TEdit`, have the name "Text", and be in the property category `TLocalizableCategory`.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, 'TEdit', 'Text');
```

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 Chapter 9, "Working with packages and components".

To create and compile a package, see "Creating and editing packages" on page 9-7. 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" on page 9-6.

Troubleshooting custom components

A common problem when registering and installing custom components is that the component does not show up in the list of components after the package is successfully installed.

The most common causes for component not showing up 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

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

Creation of every component begins the same way: you create a unit, derive a component class, register it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 31-7.

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 Component palette.

- The resulting unit should look like this:

```

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.

```

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 class

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) 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" on page 32-8.

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:

```

type
  TWrapMemo = class(TMemo)
  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;

```

Now you can install the new component on the Component 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 Chapter 38, “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*:

```

type
  TWrapMemo = class(TMemo)
  :
  published
    property WordWrap default False;
  end;

```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component’s constructor. Redefining the default ensures that Delphi knows when to write *WordWrap* to the form file.

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 a window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component is on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows 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

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 31-7.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *Shapes*.
- Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
- Register *TSampleShape* on the Samples page of the Component palette.

The resulting unit should look like this:

```
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.

```

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” on page 33-2 and “Making events visible” on page 34-5. 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:

```

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;

```

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 Chapter 33, “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.

```
type
    TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
        sstEllipse, sstCircle);
    TSampleShape = class(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 *TShape*:

```
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;
```

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:

```
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;
```

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 Chapter 39, “Modifying an existing component.”.

In this example, the shape control sets its size to a square 65 pixels on each side.

Add the overridden constructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public { constructors are always public }
    constructor Create(AOwner: TComponent); override { remember override directive }
  end;
```

1 Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
```



```

:
published
  property Height default 65;
  property Width default 65;
end;

```

2 Write the new constructor in the **implementation** part of the unit:

```

constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;

```

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:

```

type
  TSampleShape = class(TGraphicControl)
  private
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush; { a field for the brush object }
    :
  end;

```

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.

```

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;

```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```

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;

```

To directly assign the contents of *Value* to *FBrush*...

```
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:

```

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;

```

2 Add the overridden destructor to the declaration of the component class:

```

type
  TSampleShape = class(TGraphicControl)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;      { remember override directive }
  end;

```

3 Write the new destructor in the **implementation** part of the unit:

```

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;

```

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:

```

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(True);                   { erase and repaint the component }
  end;

```

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:

```
type
  TSampleShape = class(TGraphicControl)
  :
  protected
    procedure Paint; override;
  :
  end;
```

Then write the method in the **implementation** part of the unit:

```
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;
```

Paint is called whenever the control needs to update its image. Windows tells controls to paint 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:

```

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;

```


Customizing a grid

Delphi provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you will see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

The resulting component is similar to the *TCalendar* component on the Samples page of the Component palette.

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 Component palette. This process is outlined in “Creating a new component” on page 31-7.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *CalSamp*.
- Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
- Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit 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” below.

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:

```

type
    TSampleCalendar = class(TCustomGrid)
    published
        property Align; { publish properties }
        property BorderStyle;
        property Color;
        property Ctl3D;
        property Font;
        property GridLineWidth;
        property ParentColor;
        property ParentFont;
        property OnClick; { publish events }
    end;

```



```

property OnDbClick;
property OnDragDrop;
property OnDragOver;
property OnEndDrag;
property OnKeyDown;
property OnKeyPress;
property OnKeyUp;
end;

```

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 Component 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.

```

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;

```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the cells

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 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 “Creating new message handlers” on page 37-5.

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:

```

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;

```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

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:

```

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;

```

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:

```

type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  :
  :

```

- 2 Initialize the date field in the constructor:

```

constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);          { this is already here }
  :                                  { other initializations here }
  FDate := Date;                     { get current date from RTL }
end;

```

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:

```

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;

```

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:

```

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;
    :

```

2 Declare and write the implementation methods, setting different elements for each index value:

```

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;
end;

```

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:

```

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;

```

- 2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```

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;

```

- 3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```

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;

```

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:

```

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;

```

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*:

```

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;

```

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. After incrementing or decrementing *CalendarDate*, decode the date value to fill the Year, Month, and Day properties with corresponding new values.

```

procedure TCalendar.NextMonth;
begin
    DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
    DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
    DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
    DecodeDate(CalendarDate, -12), Year, Month, Day);
end;

```

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.

```

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;

```

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:

```

type
    TSampleCalendar = class(TCustomGrid)
    private
        FOnChange: TNotifyEvent;
    protected
        procedure Change; dynamic;
        :
    published
        property OnChange: TNotifyEvent read FOnChange write FOnChange;
        :

```

- 2 Write the *Change* method:

```

procedure TSampleCalendar.Change;
begin
    if Assigned(FOnChange) then FOnChange(Self);
end;

```

- 3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```

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;

```

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:

```

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;

```

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

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 Chapter 26, “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 chapter 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 calendar created in Chapter 41, “Customizing a grid”, *TSampleCalendar*. You can also use the standard calendar control on the Samples page of the Component palette, *TCalendar*.

The chapter 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

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 31-7.

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 *TDBCcalendar*, descended from *TSampleCalendar*. Chapter 41, “Customizing a grid,” shows you how to create the *TSampleCalendar* component.
- Register *TDBCcalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit DBCal;

interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Grids, Calendar;

type
    TDBCcalendar = class(TSampleCalendar)
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TDBCcalendar]);
end;

end.
```

You can now proceed with making the new calendar a data browser.

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 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 unselectable.

- 1 Add the property declaration and a **private** field to hold the value:

```
type
  TDBCcalendar = 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 TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  FReadOnly := True;                 { set the default value }
end;
```

- 2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in “Excluding blank cells” on page 41-12.

```
function TDBCcalendar.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;
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCcalendar*, 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*:

```
type
  TDBCcalendar = 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 TDBCcalendar.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 TDBCcalendar.UpdateCalendar;
begin
    FUpdating := True;                                       { set flag to allow updates }
    try
        inherited UpdateCalendar;                           { update as usual }
    finally
        FUpdating := False;                                  { always clear the flag }
    end;
end;
end;

```

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 Chapter 40, “Creating a graphic component”.

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” on page 40-5. 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:

```

type
    TDBCcalendar = class(TSampleCalendar)
    private
        FDataLink: TFieldDataLink;
    :
    end;

```

Before you can compile the application, you need to add DB and DBCtrls to the unit's `uses` clause.

Declaring the access properties

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 Chapter 40, "Creating a graphic component", 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:

An example of declaring access properties

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    ...
    function GetDataField: string;           { returns the name of the data field }
    function GetDataSource: TDataSource;     { returns reference to the data source }
    procedure SetDataField(const Value: string); { assigns name of data field }
    procedure SetDataSource(Value: TDataSource); { assigns new data source }
  published
    { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
:
function TDBCcalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCcalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCcalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCcalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

```

Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link class when the calendar control is constructed, and destroy the data link before destroying the calendar.

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 data-link object, respectively:

```

type
  TDBCcalendar = class(TSampleCalendar)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    :
  end;
:
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  FDataLink := TFieldDataLink.Create;           { construct the data-link object }
  inherited Create(AOwner);                     { always call the inherited constructor first }
  FReadOnly := True;                           { this is already here }
end;

destructor TDBCcalendar.Destroy;
begin
  FDataLink.Free;                              { always destroy owned objects first... }
  inherited Destroy;                            { ...then call inherited destructor }
end;

```

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.

```

type
  TDBCcalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);           { must have proper parameters for event }
  end;
:

constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                         { always call the inherited constructor first }
  FReadOnly := True;                                { this is already here }
  FDataLink := TFieldDataLink.Create;               { construct the data-link object }
  FDataLink.OnDataChange := DataChange;            { attach handler to event }
end;

destructor TDBCcalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;                   { detach handler before destroying object }
  FDataLink.Free;                                  { always destroy owned objects first... }
  inherited Destroy;                               { ...then call inherited destructor }
end;

procedure TDBCcalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                    { if there is no field assigned... }
    CalendarDate := 0;                             { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime; { otherwise, set calendar to the date }
end;

```

You now have a data-browsing control.

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 the first part of the chapter. 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:

```

constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  :
  FReadOnly := False; { set the default value }
  :
end;

```

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 *TDBCcalendar* class:

```

type
  TDBCcalendar = class (TSampleCalendar);
  :
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
    :
  end;

procedure TDBCcalendar.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;

```

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 *TDBCcalendar* class:

```

type
  TDBCcalendar = class(TSampleCalendar);
  :
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
  :
  end;

```

- 2 Implement the *KeyDown* method:

```

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;
end;

```

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 *TDBCcalendar* 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.

- 1 To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
    :
  end;
```

- 2 Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;      { set field link to calendar date }
end;
```

- 3 Within the constructor for *TDBCcalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

Modifying the Change method

The *Change* method of the *TDBCcalendar* 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 *TDBCcalendar* component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure Change; override;
    :
  end;
```

- 2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
TDBCcalendar.Change;
begin
  FDataLink.Modified;                               { call the Modified method }
  inherited Change;                                  { call the inherited Change method }
end;
```

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.

VCL has defined 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 Chapter 37, “Handling messages.”

To update the dataset within a message handler, follow these steps:

- 1 Add the message handler to the *TDBCcalendar* component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
    :
  end;
```

- 2 Implement the *CMExit* method so it looks something like this:

Creating a data-editing control

```
procedure TDBCcalendar.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;
```

Making a dialog box a component

You will find it convenient to make a frequently used dialog box into a component that you add to the Component palette. Your dialog box components will work just like the components that represent the standard Windows 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 chapter, 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 Component palette. This process is outlined in "Creating a new component" on page 31-7.

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 Component palette.

The resulting unit should look like this:

```
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.
```

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
- 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:

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms
  About;
```

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:

```
var
  AboutBox: TAboutBox;
```

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.

For example, to declare an integer-type interface property called *Year*, you would declare it as follows:

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;           { field to hold the Year-property data }
  published
    property Year: Integer read FYear write FYear;      { property matched with storage }
  end;
```

For this About box, you need four string-type properties—one each for the product name, the version information, the copyright information, and any comments.

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string;      { declare fields }
  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

When you install the component onto the Component palette and place the component on a form, you will be able to set the properties, and those values will automatically stay with the form. The wrapper can then use those values when executing the wrapped dialog box.

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 will 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:

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

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*.

Here is the minimal *Execute* method for a dialog-box form of type *TMyDialogBox*:

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);           { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK);             { execute; set result based on how closed }
  finally
    DialogBox.Free;                                     { dispose of the form }
  end;
end;
```

Note the use of a **try..finally** block to ensure that the application disposes of the dialog-box object even if an exception occurs. In general, whenever you construct an object this way, you should use a **try..finally** block to protect the block of code and make certain the application frees any resources it allocates.

In practice, there will be more code inside the **try..finally** block. Specifically, before calling *ShowModal*, the wrapper will set some of the dialog box's properties based on the wrapper component's interface properties. After *ShowModal* returns, the wrapper will probably set some of its interface properties based on the outcome of the dialog box execution.

In the case of the About box, you need to use the wrapper component's four interface properties to set the contents of the labels in the About box form. Because the About box does not return any information to the application, there is no need to do anything after calling *ShowModal*. Write the About-box wrapper's *Execute* method so that it looks like this:

Within the public part of the *TAboutDlg* class, add the declaration for the *Execute* method:

```
type
  TAboutDlg = class(TComponent)
  public
    function Execute: Boolean;
  end;

function TAboutBoxDlg.Execute: Boolean;
begin
  AboutBox := TAboutBox.Create(Application);           { construct About box }
  try
    if ProductName = '' then                          { if product name's left blank... }
      ProductName := Application.Title;                { ...use application title instead }
    AboutBox.ProductName.Caption := ProductName;      { copy product name }
    AboutBox.Version.Caption := Version;              { copy version info }
    AboutBox.Copyright.Caption := Copyright;          { copy copyright info }
    AboutBox.Comments.Caption := Comments;            { copy comments }
    AboutBox.Caption := 'About ' + ProductName;       { set About-box caption }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon; { copy icon }
      Result := (ShowModal = IDOK);                   { execute and set result }
    end;
  finally
    AboutBox.Free;                                     { dispose of About box }
  end;
end;
```

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 Component 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:

```
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.

Developing COM-based applications

The chapters in “Developing COM-based applications” present concepts necessary for building COM-based applications, including Automation controllers, Automation servers, ActiveX controls, and MTS applications.

Note Support for Automation controllers is available in all editions of Delphi. However, to create servers, you need the Professional or Enterprise edition.

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 objects, Automation servers and clients (controllers), Active Server Pages, ActiveX controls, or ActiveForms.

COM is a language independent software component model designed by Microsoft to enable interaction between software components and applications. Microsoft extends this technology with ActiveX, which is primarily used for Intranet development.

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 to those features.

Applications can access COM components and their interfaces that exist on the same computer as the application, or that exist on another computer on the network using a mechanism called Distributed COM or DCOM. For more information on clients, servers, and interfaces see, "Parts of a COM application," on page 44-2.

This chapter provides a conceptual overview of the underlying technology on which Automation and ActiveX controls are built. Later chapters provide details on creating Automation objects and ActiveX controls in Delphi.

COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines object creation and communication between objects. 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 the COM library (including OLE32.dll and OLEAut32.dll), 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.

Note Delphi's interface objects and language follow the COM specification. Delphi's implementation of the COM specification is called the Delphi ActiveX framework (DAX). Most of the implementation is found in the AxCtrls unit.

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 are found in the AxCtrls unit.

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 Server Pages, and Active Documents. For details, see "COM extensions" on page 44-8.

In addition, you can create a COM object that can work within the Microsoft Transaction Server (MTS) environment. MTS is a component-based transaction processing system for building, deploying, and managing large intranet and Internet server applications. Even though MTS is not architecturally part of COM, it is designed to extend the capabilities of COM in a large, distributed environment. For more information on MTS, see Chapter 51, "Creating MTS objects."

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see "Implementing COM objects with wizards" on page 44-16

Parts of a COM application

When implementing a COM application, you supply the following:

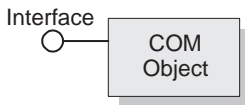
COM Interface The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods (member functions) and properties (data members and or content).

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. The most common COM client to implement is an Automation controller. Delphi eases the process in creating a client by allowing you to install COM servers (such as a Word document or Powerpoint slide) as components on the Component 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 shown in Figure 44.1:

Figure 44.1 A COM interface



For example, every COM object implements the basic interface, *IUnknown*, which tells the client what interfaces are available on the client.

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.
- 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,” on page 44-6.

The fundamental COM interface, IUnknown

All COM objects must support the fundamental interface, called *IUnknown*, which contains the following routines:

<i>QueryInterface</i>	Provides pointers to other interfaces that the object supports.
<i>AddRef</i> and <i>Release</i>	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.

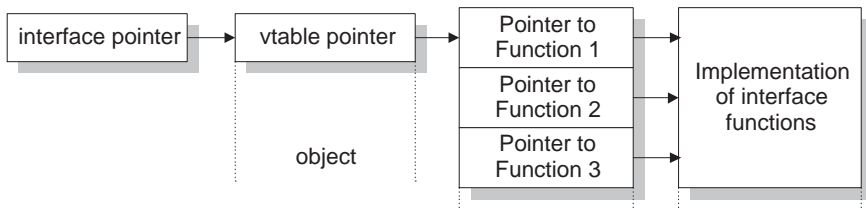
Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface*. *QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any 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 32-bit 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 ObjectPascal.

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.

Figure 44.2 Interface vtable

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 (data members or content) and methods (or member functions).

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 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 references a COM object. COM uses this CLSID to locate the appropriate server implementation, brings the code into memory, and has the server instantiate an object instance for the client. Thus, a COM server must provide a class factory object (*IClassFactory*) that creates instances of objects on demand. (The CLSID is based on the interface's GUID.)

In general, 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 is a special type of object that 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 as described in "Implementing COM objects with wizards" on page 44-16.

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 CoClass 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 and registers an object instance for that particular client. If another client requests the object's services, the class factory creates another object instance to service the second client.

A CoClass must have a class factory and a class identifier (CLSID) so that its COM object 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 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.

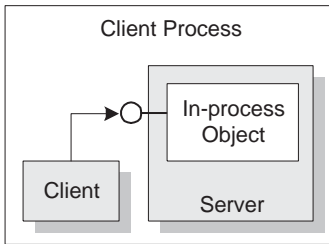
Out-of-process server (or local server) Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine.

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.

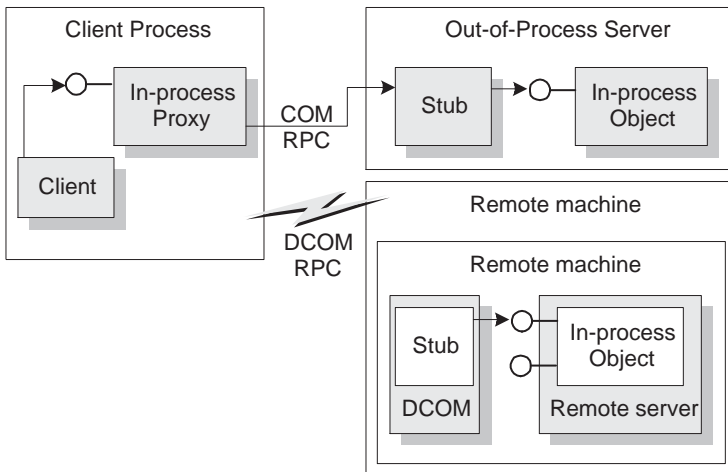
The remote server uses distributed COM (DCOM) interfaces to communicate with the application server.

As shown in Figure 44.3, 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.

Figure 44.3 In-process server

As shown in Figure 44.4, 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.

Figure 44.4 Out-of-process and remote servers

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.

What type of marshaling occurs depends on what 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 Chapter 47, "Creating an Automation server."

Note Microsoft Transaction Server (MTS) provides additional support for remote objects. For details, see Chapter 51, "Creating MTS objects."

COM clients

It is important to design a COM application where clients can query the interfaces of an object to determine what an object is capable of providing. Server objects should have no expectations about the client using its objects. The client can determine what an object can provide through its interfaces. In addition, clients don't need to know how (or even where) an object provides the services; it just needs to rely on the object to provide the service it advertised through the interface.

A typical COM client is the Automation Controller. An Automation controller is the part of the application that has the broad perspective of the application's intended use. It knows the kinds of information it needs from various objects on the server, and it requests services when necessary.

Delphi makes it easier for you to develop an Automation Controller by allowing you to import an Automation server's type library and install it on the Component palette.

For details in creating an Automation controller, see Chapter 46, "Creating an Automation controller."

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.

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.

Microsoft is incorporating some of the MTS technologies for building complex Internet and intranet applications within the COM framework. This next evolution of

COM, which also incorporates other new features, is currently called “COM+” (COM Plus) and should be available with the release of Windows 2000.

The following lists the vast array of services COM extensions currently provide. Subsequent sections describe these services in greater detail.

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.

Automation controllers (or COM clients) Clients of Automation servers. Controllers provide a programming environment in which a developer or user can write scripts (controls) to drive Automation servers.

ActiveX controls ActiveX controls are specialized in-process COM servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.

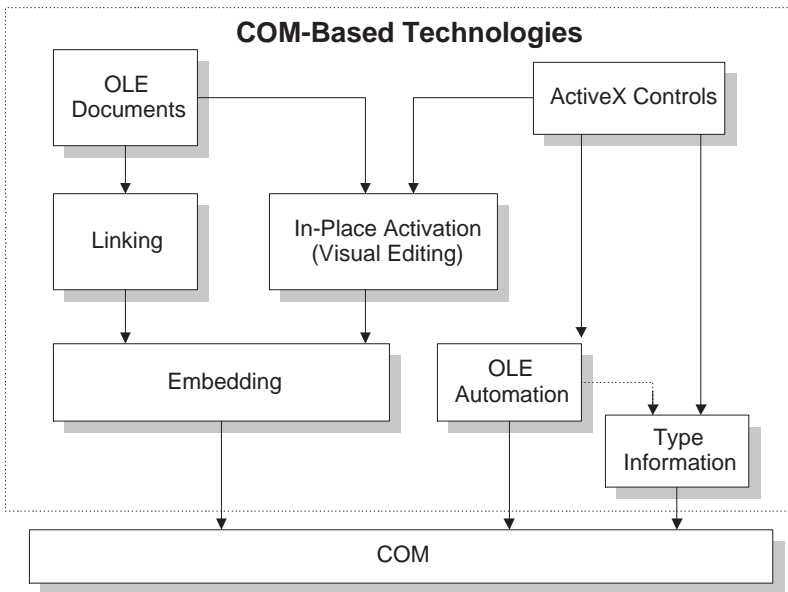
Type libraries A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers and ActiveX controls expect type information to be available.

Active Server Pages Active Server Pages are ActiveX components that allow you to create dynamic Web pages.

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.

Visual cross-process objects Objects that can be manipulated across different processes.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:

Figure 44.5 COM-based technologies

Using COM objects introduces both features and restrictions. These 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.

Table 44.1 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.

Table 44.1 COM object requirements

Object	Visual Object?	Process space	Communication	Type library
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the <i>IDispatch</i> interface (for out-of-process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the <i>IDispatch</i> interface	Required
Custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
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 and controllers

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 client of an Automation object is referred to as an Automation controller, and the server object being manipulated is called the Automation object.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object must be able to define a set of properties and commands, and to describe their capabilities through type descriptions. In order to do this they must have a way to provide information about the object's interfaces, the interface methods, and those methods' arguments. Typically this information is available in type libraries. The Automation server can also generate type information dynamically when queried.
- Automation objects must make these methods accessible so that other applications can use them. For this, they must 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 the section on Valid Types in Chapter 50, "Working with type libraries."

For details on writing an Automation controller, see Chapter 46, "Creating an Automation controller." For information on writing an Automation server, see Chapter 47, "Creating an Automation server."

ActiveX controls

ActiveX controls are visual controls that run only in 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 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.

An increasingly common use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX has become a standard that has especially targeted 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.

Delphi wizards allow you to easily create ActiveX controls. For more information about creating and using these types of objects, see Chapter 48, “Creating an ActiveX control.”

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 and ActiveX controls expect type information to be available. Automation and ActiveX wizards generate a type library automatically. You can view or edit this type information by using the Type Library Editor as described in Chapter 50, “Working with type libraries.”

This section describes what a type library contains, how it is created, when it is used, and how it is accessed. For developers wanting to share interfaces across languages, the section ends with suggestions on using type library tools.

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 for you when you use either the Automation server or ActiveX control wizard. (You can also create a type library by choosing from the main menu,

File | New | ActiveX | Type Library.) Then 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 appropriate source files when the type library is saved.

The Type Library editor automatically generates a standard type library, typically as a resource, along with a Delphi Interface File (.PAS file) that contains the type declarations in Object Pascal syntax. For more information on using the Type Library editor to write interfaces and CoClasses, see Chapter 50, "Working with type libraries."

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. For more information about vtable and compile time binding, see "Creating an Automation object for an application" on page 47-1.
- 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 *IProvideClassInfo* 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. Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Interface	Description
<i>ITypeLib</i>	Provides methods for accessing a library of type descriptions.
<i>ITypeInfo</i>	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.
<i>ITypeComp</i>	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. 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 (as opposed to calling through Variants), and controllers that do not support vttables 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, is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- MKTYPLIB is an ODL compiler that compiles ODL scripts to create a type library, found in the MS Win32 SDK.
- OLEView is a type library browser tool, found on Microsoft's Web site.
- 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 Examples directory.
- RegSvr32.exe is a tool for registering and unregistering servers and type libraries, which is a standard Windows utility.

Active Server Pages

The Active Server Pages (ASP) technology allows you to build Web pages dynamically by using ActiveX server components. With Active Server Pages, you can

embed ActiveX controls in a Web page that get called every time the server loads the Web page. For example, you can write an Object Pascal program, such as one to create a bitmap or connect to a database, and this control accesses data that gets updated every time the server loads the Web page.

Active Server Pages relies on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

On the server side, the Active Server Pages are ActiveX components which you can develop using Delphi or most other languages including C++, Java, or Visual Basic. On the client side, the ASP is a standard HTML document and can be viewed by users on any platform using any Web browser.

Delphi wizards allow you to easily create Active Server Pages. For more information about creating and using these types of objects, see Chapter 49, "Creating an Active Server Page."

Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that supports linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, for example, 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. So, 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 in existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use one of the VCL COM base classes and implement the appropriate interfaces for that object type, 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 for 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, for example, window handles, menu handles, and so on.

Visual cross-process objects

Automation objects, Active Documents and ActiveX controls are commonly used objects. Less common are OLE/ActiveX objects that are visually displayed and

manipulated in a cross-process application. These types of objects are more difficult to create because the communication protocol involved in visually manipulating objects in cross-process applications is only standardized for visual objects that use Active Document interfaces. Therefore, you need to write one or more custom interfaces that your object implements, and then handle the marshaling interfaces yourself.

This can be done by either

- Using a dual interface *IDispatch*, which provides automatic marshaling. This is the recommended way. It is the easiest approach, and the Automation wizard creates dual interfaces by default when you create an Automation object.
- Writing the marshaling classes by hand by implementing *IMarshal* or related interfaces.

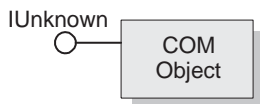
Implementing COM objects with wizards

Delphi makes it easier to write COM applications by providing wizards to help create Delphi applications that run in the COM environment. It provides separate wizards to create the following:

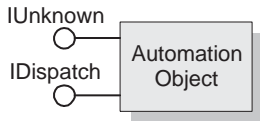
- A simple COM object
- An Automation object
- An ActiveX control
- An Active server page
- An ActiveX Form
- ActiveX library
- Property page
- Type library
- MTS object

The wizards automate the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. As shown in Figure 44.6, with a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

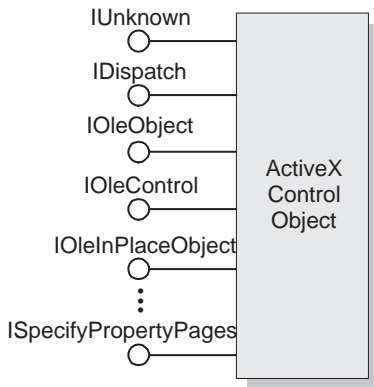
Figure 44.6 Simple COM object interface



As shown in Figure 44.7, for Automation objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.

Figure 44.7 Automation object interface

As shown in Figure 44.8, for ActiveX control objects, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IObject*, *IOleControl*, and so on. For a complete list of interfaces, see the reference page for *TActiveXObject*.

Figure 44.8 ActiveX object interface

You can think of the wizards as providing an increasing degree of implementation. As shown in Table 44.2, the various wizards implement the following COM interfaces:

Table 44.2 Delphi wizards for implementing COM, Automation, and ActiveX objects

Wizard	Implemented interfaces	What the wizard does
COM server	<i>IUnknown</i>	<p>Exports the necessary routines that handle server registration, class registration, loading and unloading the server, and object instantiation.</p> <p>Creates and manages the class factories for objects implemented on the server.</p> <p>Instructs COM to invoke object interfaces based on a specified threading model.</p> <p>Provides a type library, if requested.</p>
Automation server	<i>IUnknown</i> , <i>IDispatch</i>	<p>Performs the tasks of a COM server wizard (described above), plus:</p> <p>Implements the interface that you specify, either dual or dispatch.</p> <p>Provides a type library automatically.</p>

Table 44.2 Delphi wizards for implementing COM, Automation, and ActiveX objects (continued)

Wizard	Implemented interfaces	What the wizard does
ActiveX Control	<i>IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages</i>	<p>Performs the tasks of the COM server and Automation server wizards (described above), plus:</p> <p>Implements the properties, methods, and events for all the interfaces in the TActiveXControl.</p> <p>Leaves you in the source code editor so that you can modify the object.</p>
ActiveForm	Same interfaces as ActiveX Control	<p>Performs the tasks of the ActiveX control wizard, plus:</p> <p>Implements the properties, methods, and events for all the interfaces in the TActiveXControl.</p> <p>Leaves you with a form so that you can design an application.</p>
Active Server Object	<i>IUnknown, IDispatch</i>	<p>Performs the tasks of an Automation object wizard (described above) and generates an .ASP page which can be loaded into a Web browser. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed.</p> <p>If you set the Active Server Type to Page-Level event methods, it implements <i>OnStartPage</i> and <i>OnEndPage</i> automatically for you.</p>
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.
Property Page	<i>IUnknown, IPropertyPage</i>	Creates a new property page that you can design in the Forms designer.
Type Library	None, by default	Creates a new type library and associates it with the active project.
MTS object	The <i>IObjectControl</i> interface methods, <i>Activate</i> , <i>Deactivate</i> , and <i>CanBePooled</i> .	Adds a new unit to the current project containing the MTS object definition, so that clients can access this server within the MTS runtime environment. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed.

If you want to add any additional COM objects (or reimplement any existing implementation), you can do so. To provide a new interface, you would create a descendant of the *IDispatch* interface and implement the needed methods in that descendant. To reimplement an interface, create a descendant of that interface and modify the descendant.

Creating a simple COM object

Delphi provides wizards to help you create various COM objects. This chapter provides an overview of how to create a simple, lightweight COM object, such as a shell extension, in the Delphi environment. To create Automation client and server objects, see Chapter 46, “Creating an Automation controller,” and Chapter 47, “Creating an Automation server.” To create an ActiveX control, see Chapter 48, “Creating an ActiveX control.” To create an Active Server Page, see Chapter 49, “Creating an Active Server Page.”

This chapter is not intended to provide complete details of writing COM applications. For that information, refer to your Microsoft Developer’s Network (MSDN) documentation. The Microsoft Web site also provides current information on this topic.

Overview of creating a COM object

Use the COM object wizard to create a simple, lightweight COM object, for example, a shell extension. A COM object can be implemented as either an in-process server, out-of-process server, or remote server.

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.

The process of creating a COM object involves these steps:

- 1 Design the COM object.
- 2 Use the COM object wizard to create a COM object.
- 3 Register the COM object.
- 4 Test the COM object.

Designing a COM object

When designing the COM object, you need to decide what COM interfaces you want to implement. The wizard supplies the *IUnknown* interface. If you want to implement any other COM interfaces, see the MSDN documentation.

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 using a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers.

For information on server types, see, “In-process, out-of-process, and remote servers,” on page 44-6.

Creating a COM object with the COM object wizard

Before you create a COM object, create or open the project for an 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 to open the New Items dialog box.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the COM object icon.

In the wizard, specify the following:

ClassName	Specify the name for the object that you want to implement.
Instancing	Specify an instancing mode to indicate how your COM object is launched. See “COM object instancing types” on page 45-3 for details. Note: When your COM object is used only as an in-process server, instancing is ignored.
Threading model	Choose the threading model to indicate how client applications can call your COM object’s interface. For details, see “Choosing a threading model” on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Implemented interfaces	Specify the names of the COM interfaces that you want this COM object to implement.
Description	Enter a description of the COM object you are creating.

Include Type Library	Check this box to generate a type library for this object. A type library contains type information that allows you to expose any object interface and its methods and properties to client applications. Checking this box automatically causes Mark Interface OleAutomation to be checked.
Mark interface OleAutomation	Check this box to enable the marshaling code that gets generated when you create a type library. COM knows how to marshal all the Automation-compatible types in the type library and can set up the proxies and stubs for you so that you can pass parameters to out-of-process (.EXE) servers. For more information, see “The marshaling mechanism” on page 44-7.

COM object instancing types

Note Instancing is ignored when your COM object is used only as an in-process server. When your COM application 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. For example, a word processor application may have a document object that can only be created by calling a method of the application that can create the document object.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in creating multiple applications. Single instance specifies that once an application has connected to the object, it is removed from public view so that no other applications can connect to it. This option is commonly used for multiple document interface (MDI) applications. When a client requests services from a single instance object, all requests are handled by the same server. For example, any time a user requests to open a new document in a word processor application, typically the new document opens in the same application process.
Multiple Instance	Specifies that multiple applications can connect to the object. Any time a client requests service, a separate instance of the server gets invoked. (That is, there can be multiple instances in a single executable.) Any time a user attempts to open the Windows Explorer, a separate Explorer is created.

Choosing a threading model

When creating an object from the 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.

Table 45.1 lists the different threading models you can specify.

Table 45.1 Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	No thread support. Client requests are serialized by the calling mechanism.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	Clients can call an object's methods only from the thread on which the object was created. Different objects from the same server can be called on different threads, but each object is called only from that one thread.	Instance data is safe, 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. Objects are easy to write, but clients can be tricky. Primarily used for controls for Web browsers.
Free (also called multi-threaded apartment)	Clients can call any object's methods from any thread at any time. Objects can handle 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 <i>not</i> reliable across multiple calls. Clients are easy to write, but objects are harder to write. Primarily used for distributed DCOM environments.
Both	Objects can support clients that use either apartment or free threading models.	Maximum performance and flexibility. Supports both threading models when clients may be either using single-threaded or free for improved performance.

Both the client and server sides of the application tell COM the rules it intends to follow for using threads when it initializes COM. COM compares the threading rules that each party has agreed to support. If both parties support the same rules, COM sets up a direct connection between the two and trusts that the parties follow the rules. If the two parties advertise different rules, COM sets up marshaling between the two parties so that each sees only the threading rules that it says it knows how to handle. Of course, marshaling affects performance somewhat, but it allows parties with different threading models to work together.

Note The threading model you choose in the wizard determines how the object is advertised in the registry. You must make sure that your object implementation adheres to the threading model you have chosen.

The threading model is valid for in-process servers only. Setting the threading model in the wizard sets the threading model key in the CLSID registry entry, `InProcessServer32`.

Out-of-process servers are registered as EXE and it is up to you to implement the threading model yourself. If the EXE contains a free threaded server, Delphi will initialize COM 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, see the `CoInitFlags` variable in online Help.

Note Local variables 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.

Writing an object that supports the free threading model

Use the free threading 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.

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 interface's methods may only be called from 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.

For general information on threads, see Chapter 8, "Writing multi-threaded applications."

Registering a COM object

After you have created your COM object, you must register it so that other applications can find and use it.

Note Before you remove a COM object from your system, you should unregister it.

To register a COM object,

- Choose Run | Register ActiveX Server.

To unregister a COM object,

- Choose Run | Unregister ActiveX Server.

As an alternative, you can use the **regsvr** command from the command line or run the **regsvr32.exe** from the operating system.

Testing a COM object

Testing your COM object depends on the COM object you have designed. Once you have created the COM object, test it by using the interfaces you implemented to access the methods of the interfaces.

To test and debug a COM 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 | Debugger Options dialog.
- 2 For an in-process server, choose Run | Parameters.
- 3 In the Host Application box, type the name of the client application which will be requesting the service of this COM object, and choose OK.
- 4 Choose Run | Run.
- 5 Set breakpoints in the COM object.
- 6 Use the client application to interact with the COM object.

The COM object pauses when the breakpoints are reached.

Creating an Automation controller

Automation is a COM protocol that defines how one application accesses an object that resides within another application or DLL. An *Automation controller* is a client application that controls an Automation server through one or more server-provided objects which implement the *IDispatch* interface.

Automation controllers can be written in any language for which an implementation of COM and Automation is provided. Most automation controllers are currently written in C++, Object Pascal (Delphi), or Visual Basic.

Examples of Automation servers include Microsoft Word, Microsoft Excel, and Internet Explorer. These applications can be controlled by Delphi applications or by other Automation controllers.

Delphi gives you the flexibility to integrate applications and DLLs with a variety of applications as either Automation servers or as controllers.

This chapter describes how to create an Automation controller in the Delphi environment. It is not intended to provide complete details of controller application development for every type of server. For specific information on a server application, you should consult that application's documentation.

For information about creating an Automation server, see Chapter 47, "Creating an Automation server."

In Delphi, you create an Automation controller by importing an Automation server's type library and installing it on the Component palette.

Creating an Automation controller by importing a type library

You can create an Automation controller by importing an Automation server's type library and using the automatically generated classes to control the Automation server. With the Import Type Library dialog, you install the server that the type library describes to the Component palette, which allows you to connect to the server

and hook up its events using the Object Inspector. You can then manipulate the server properties programmatically.

To import an Automation controller's type library,

- 1 Choose Project | Import Type Library.
- 2 Select the type library from the list.

The dialog lists all the libraries registered on this system. If the type library is not in the list, choose the Add button, find and select the type library file, choose OK, and repeat step 2. Note that the type library could be a standalone type library file (.TLB, .OLB), or a server that provides a type library (.DLL, .OCX, .EXE).

- 3 Choose the Palette page on which this server will reside.
- 4 Check Generate Component Wrapper, which creates a *TComponent* wrapper that allows you to install the server that the type library describes on the Component palette.
- 5 Choose Install.

Specify where you want the type library to reside, either in an existing package or new package. This button is grayed out if no component can be created for that type library.

A server that the type library described now resides on the Component palette. You can use the Object Inspector to write an event handler for the server.

To control the Automation server through this controller, you add code to the implementation unit to provide support for early (VTable) binding and/or late (dispatch) binding. The unit that you just created includes interface wrappers for both VTable binding for all exposed interfaces, and dispatch binding for dual and dispatch interfaces.

Handling events in an automation controller

Once you have installed a server on the Component palette, you can use the Object Inspector to write an event handler.

To support events,

- 1 From the Component palette, drop the desired server component onto your form.
- 2 Select an object, and click the Events tab on the Object Inspector. You'll see a list of the object's events.
- 3 Double-click in the space to the right of an event and Delphi opens the Code editor, displaying the skeleton event handler for you to complete.

After implementing your event handler, you can connect to a server.

Connecting to and disconnecting from a server

Typically, you connect to a server through its main interface. For example, you would connect to Microsoft Word through the `WordApplication` component. Once you've connected to the main interface, you can connect to any of the application's components (such as a `WordDocument` or `WordParagraphFormat`) by using the `ConnectTo` method.

Here is the event handler for the `ExcelApplication` event, `OnNewWorkbook`. In it, we are assigning the workbook to the `ExcelWorkbook` component.

```
procedure TForm1.XLAppNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
    ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
```

After importing the type library, add code to the implementation unit to control the server with either a dual interface (most typical) or a dispatch interface.

Note The import files created when a type library is imported (`libname_TLB.CPP` and `libname_TLB.H`) should be considered to be read-only, and are not intended for modification of any kind. These files are regenerated each time the type library is refreshed, so any changes are overwritten.

For servers that expose a `Quit` method (such as `WordApplication` and `ExcelApplication`), code is generated to call this method in the `Disconnect` method. `Quit` typically exposes functionality that is equivalent to clicking on `File` to quit the application. If `AutoConnect` is set to `True`, the application server will `Quit` when the client exits. Therefore, hitting `F1` when `AutoQuit` is highlighted in the `Object Inspector` does nothing.

Controlling an Automation server using a dual interface

When you create an Automation controller by selecting an object from the `Component Palette`, it automatically provides a dual interface because Delphi can determine the `VTable` layout from information in the type library. Calling a method on the class automatically connects to an instance of `Word`.

For example, for a `Word` server, you call on a method of class `TWordApplication` as follows:

```
foo := TWordapplication {New way}
foo.DoSomething;
```

Of course, the former way of controlling an Automation server with a dual interface still works, but it is more cumbersome. You must first, declare an interface and initialize it with the `Create` method of a `CoClass` client proxy class. Then you call methods of the smart interface object. For example,

```
foo : _Application
foo := CoWordApplication.Create {Old way}
foo.DoSomething;
```

The interface and CoClass client proxy class are defined in the unit that is generated automatically when you import a type library. The names that start with “I” are smart interfaces. For example, the main smart interface for Microsoft Word is “IWordBasic”. (A smart interface automatically frees its wrapped interface when it is destroyed.) The names of the CoClass client proxy classes start with “Co”. For example, the main CoClass client proxy class for Microsoft Word is CoApplication. For information about dual interfaces, see “Automation interfaces” on page 6.

Controlling an Automation server using a dispatch interface

Typically, you will use the dual interface to control the Automation server, as described above. However, you may find a need to control an Automation server with a smart dispatch interface object. To do so,

- 1 In the Automation controller’s implementation unit, declare a dispinterface.
- 2 Control the Automation server by calling methods of the dispatch interface object.

For information on dispatch interfaces, see “Automation interfaces” on page 47-6.

Example: Printing a document with Microsoft Word

The following steps show how to create an Automation controller that prints a document using Microsoft Word 8 from Office 97.

Before you begin: Create a new project that consists of a form, a button, and an open dialog box (*TOpenDialog*). These controls constitute the Automation controller.

Step 1: Prepare Delphi for this example

For your convenience, Delphi has provided many common servers, such as Word, Excel, and Powerpoint, on the Component Palette. To demonstrate how to import a server, we use Word. Since it already exists on the Component Palette, this first step asks you to remove the package containing Word so that you can install it on the palette. Step 4 describes how to return the Component Palette to its normal state.

To remove Word from the Component palette,

- 1 Choose Component | Install packages.
- 2 Click Borland Sample Automation Server components and choose Remove.

The Servers page of the Component Palette no longer contains any of the servers supplied with Delphi. (If no other servers have been imported, the Servers page also disappears.)

Step 2: Import the Word type library

To import the Word type library,

- 1 Choose Project | Import Type Library.

- 2 In the Import Type Library dialog,
 - 1 Select Microsoft Office 8.0 Object Library.

If Word (Version 8) is not in the list, choose the Add button, go to Program Files\Microsoft Office\Office, select the Word type library file, MSWord8.olb choose Add, and then select Word (Version 8) from the list.
 - 2 In Palette Page, choose Servers.
 - 3 Choose Install.

The Install dialog appears. Select the Into New Packages tab and type WordExample to create a new package containing this type library.
- 3 Go to the Servers Palette Page, select WordApplication and place it on a form.
- 4 Write an event handler for the button object as described next.

Step 3: Use a VTable or dispatch interface object to control Microsoft Word

You can use either a VTable or a dispatch object to control Microsoft Word.

Using a VTable interface object

By dropping an instance of the WordApplication object onto your form, you can easily access the control using a VTable interface object. You simply call on methods of the class you just created. For Word, this is the TWordApplication class.

- 1 Select the button, double-click its *OnQuit* event handler and supply the following event handling code:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  FileName: OleVariant;
begin
  if OpenFileDialog1.Execute then
    begin
      FileName := OpenFileDialog1.FileName;

      WordApplication1.Documents.Open(FileName,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam);

      WordApplication1.ActiveDocument.PrintOut(
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam);
    end;
end;

```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Using a dispatch interface object

As an alternate, you can use a dispatch interface for late binding. To use a dispatch interface object, you create and initialize the Application object using the `_ApplicationDisp` dispatch wrapper class as follows. Notice that dispinterface methods are “documented” by the source as returning Vtable interfaces, but, in fact, you must cast them to dispatch interfaces.

- 1 Select the button, double-click its `OnQuit` event handler and supply the following event handling code:

```

procedure TForm1.Button1Click(Sender: TObject);

var
  MyWord : _ApplicationDisp;
  FileName : OleVariant;
begin
  if OpenDialog1.Execute then
    begin
      FileName := OpenDialog1.FileName;
      MyWord := CoWordApplication.Create as
        _ApplicationDisp;
      (MyWord.Documents as
        DocumentsDisp).Open(FileName, EmptyParam,
          EmptyParam, EmptyParam, EmptyParam, EmptyParam,
          EmptyParam, EmptyParam, EmptyParam, EmptyParam);
      (MyWord.ActiveDocument as
        _DocumentDisp).PrintOut(EmptyParam, EmptyParam,
          EmptyParam, EmptyParam, EmptyParam, EmptyParam,
          EmptyParam, EmptyParam, EmptyParam, EmptyParam,
          EmptyParam, EmptyParam, EmptyParam, EmptyParam);
      MyWord.Quit(EmptyParam, EmptyParam, EmptyParam);
    end;
end;

```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Step 4: Clean-up the example

After completing this example, you will want to restore Delphi to its original form.

- 1 Delete the objects on this Servers page:
 - 1 Choose Component | Configure Palette and select the Servers Page.
 - 2 Select all objects on the page that you just added, choose Hide.
 - 3 The Servers page no longer appears.

- 2 Return the Borland Sample Automation Server Components package:
 - 1 Choose Component | Install Packages.
 - 2 From the list, select the WordExample package and click remove.
 - 3 From the list, select Borland Sample Automation Server Components package, and click Add.
The Servers tab is returned to the Component Palette.

Getting more information

For the most up-to-date information about the dispatch interface, dual interface, and CoClass client proxy classes, refer to the comments in the automatically-generated source file.

Creating an Automation server

An Automation server is an application that exposes its functionality for client applications, called Automation controllers, to use. Controllers can be any applications that support Automation, such as Delphi, Visual Basic, or C++Builder. An Automation server can be an application or library.

This chapter shows how to create an Automation server using the Delphi Automation server wizard. With this, you can expose properties and methods of an existing application for Automation control.

Here are the steps for creating an Automation server from an existing application:

- Create an Automation object for the application.
- Expose the application's properties and methods for Automation.
- Register the application as an Automation server.
- Test and debug the application.

For information about creating an Automation controller, see Chapter 46, "Creating an Automation controller." For general information about the COM technologies, see Chapter 44, "Overview of COM technologies."

Creating an Automation object for an application

An Automation object is an ObjectPascal class descending from *TAutoObject* that supports OLE Automation, exposing itself for other applications to use. Since *TAutoObject* supports OLE Automation, any object derived from it gets Automation support automatically. You create an Automation object using the Automation Object wizard.

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.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Automation Object icon.

In the wizard dialog, specify the following:

CoClass Name	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)
Instancing	Specify an instancing mode to indicate how your Automation server is launched. For details, see “COM object instancing types” on page 45-3. Note: When your Automation object is used only as an in-process server, instancing is ignored.
Threading Model	Choose the threading model to indicate how client applications can call your object’s interface. This is the threading model that you commit to implementing in the Automation object. For more information on threading models, see “Choosing a threading model” on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Generate event support code	Check this box to tell the wizard to implement a separate interface for managing events of your Automation object.

When you complete this procedure, a new unit is added to the current project that contains the definition for the Automation object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods of the interface through the type library as described next.

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. For more information, see “Dual interfaces” on page 47-6.

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 events, it provides a definition of an outgoing interface which is implemented by a client. The client determines what outgoing interfaces are available by querying the server’s *IConnectionPointContainer* interface, and it uses methods provided by the server’s *IConnectionPoint* interface to pass the server a

pointer to the client's implementation of the events (known as a sink). The server must maintain a list of such sinks and call methods on them when an event occurs. When you select *Generate Event Support Code*, Delphi automatically generates the code necessary to support *IConnectPoint* and *IConnectPointContainer*.

Exposing an application's properties, methods, and events

When you build the Automation server with the Automation wizard, it automatically generates a type library, which provides a way for host applications to find out what the object can do. To expose the properties, methods, and events of an application, you modify the Automation server's type library as described below.

Exposing a property for Automation

A property is a member function that sets or returns information about the state of the object, such as color or font. For example, a button control might have a property declared as follows:

```
property Caption: WideString;
```

To expose a property for Automation,

- 1 In the type library editor, select the default interface for the Automation object.
The default interface should be the name of the Automation object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."
- 2 To expose a read/write property, click the Property button on the toolbar; otherwise, click the arrow next to the Property button on the toolbar, and then click the type of property to expose.
- 3 In the Attributes pane, specify the name of the property.
- 4 In the Parameters pane, specify the property's return type and add the appropriate parameters.
- 5 On the toolbar, click the Refresh button.
A definition and dummy implementation for the property is inserted into the Automation object's unit files.
- 6 In the dummy implementation for the property, add code (between **try** and **finally** statements) that provides the expected functionality. In many cases, this code will simply call an existing function inside the application.

Exposing a method for Automation

A method can be a procedure or a function. To expose a method for Automation,

- 1 In the Type Library editor, select the default interface for the Automation object.

The default interface should be the name of the Automation object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."

- 2 Click the 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 toolbar, click the Refresh button.

A definition and dummy implementation for the method is inserted into the Automation object's unit files.

- 6 In the dummy implementation for the method, add code between **try** and **finally** statements that provides the expected functionality. In many cases, this code will simply call an existing function inside the application.

Exposing an event for Automation

To expose an event for Automation,

- 1 In the Automation wizard, check the box, Generate event support code.
The wizard creates an Automation object that includes an Events interface.
- 2 In the Type Library editor, select the Events interface for the Automation object.
The Events interface should be the name of the Automation object preceded by the letter "I" and succeeded by the word "Events."
- 3 Click the Method button from the Type Library toolbar.
- 4 In the Attributes pane, specify the name of the method, such as MyEvent.
- 5 On the toolbar, click the Refresh button.

A definition and dummy implementation for the event is inserted into the Automation object's unit files.

- 6 In the Code Editor, create an event handler inside the *TAutoObject* descendant in the Automation object class. For example,

```
unit ev;
interface
uses
  ComObj, AxCtrls, ActiveX, Project1_TLB;
type
  TMyAutoObject = class (TAutoObject, IConnectionPointContainer, IMyAutoObject)
  private
    .
    .
    .
  public
    procedure Initialize; override;
    procedure EventHandler; { Add an event handler}
```

- 7 At the end of the *Initialize* method, assign an event to the event handler you just created. For example,

```
procedure TMyAutoObject.Initialize;
begin
  inherited Initialize;
  FConnectionPoints:= TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint (AUtoFactory.EventIID,
      ckSingle, EventConnect);
  OnEvent = EventHandler;{ Assign an event to the event handler }
end;
```

- 8 Add the necessary code to call the method implemented by the sink. For example, provide the following code substituting the name of your event for “MyEvent.”

```
procedure TMyAutoObject.EventHandler;
begin
  if FEvents <> nil then FEvents.MyEvent;{ Call method implemented by the sink.}
end;
```

Getting more information

Press F1 anywhere in the Type Library Editor to get more information on using the editor.

Registering an application as an Automation server

You can register the Automation server 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,” on page 44-6.

Note When you want to remove the Automation server from your system, it is recommended that you 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 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.

Testing and debugging the application

To test and debug an Automation server,

- 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 | 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.

Automation interfaces

Delphi wizards implement the 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**.

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.

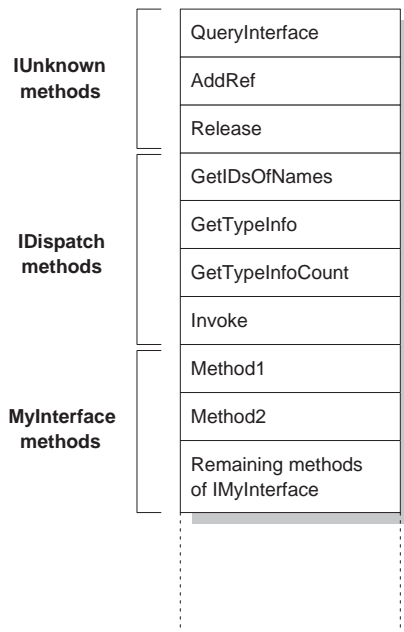
Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- 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. For more information on marshaling, see, “Marshaling data,” on page 47-8.

The following diagram depicts the *IMyInterface* interface in an object that supports a dual interface named *IMyInterface*. 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.

Figure 47.1 Dual interface VTable



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 dispinterface wrappers can avoid calls to *GetIDsOfNames*; this information is already in the `_TLB` file. 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, by 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**.

Automation compatible types

Function result and parameter types of the methods declared in dual and dispatch interfaces must be *Automation-compatible* types. The following types are OLE Automation-compatible:

- The predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see “Valid types” on page 50-23.
- 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.
- Interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.
- Dispinterface types defined in a type library.
- *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, the following restrictions apply. When you edit your Automation object using the type library editor, the editor enforces these restrictions:

- Types must be compatible for cross-platform communication. For example, you cannot use data structures (other than implementing another property object), unsigned arguments, wide strings, and so on.
- String data types must be transferred as BSTR. PChar and AnsiString cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function’s return value.
- 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 will use automatic marshaling in your 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.

Creating an ActiveX control

An ActiveX control is a software component that integrates into and extends the functionality of any host application that supports ActiveX controls, such as C++Builder, Delphi, Visual dBASE, Visual Basic, Internet Explorer, and Netscape Navigator.

For example, Delphi comes with several ActiveX controls, including charting, spreadsheet, and graphics controls. You can add these controls to the component palette in the IDE, and then use them like any standard VCL component, dropping them on forms and setting their properties using the Object Inspector.

An ActiveX control can also be deployed on the Web, allowing it to be referenced in HTML documents and viewed with ActiveX-enabled Web browsers.

This chapter provides an overview of how to create an ActiveX control in the Delphi environment. It is not intended to provide complete implementation details of writing ActiveX control. For that information, refer to your Microsoft Developer's Network (MSDN) documentation or search the Microsoft Web site for ActiveX information.

Overview of ActiveX control creation

Delphi provides two wizards for ActiveX development:

- The ActiveX Control wizard allows you to convert an existing VCL or custom-VCL control into an ActiveX control by adding an ActiveX class wrapper around the VCL control.
- The ActiveForm wizard allows you to create a new ActiveX application from the start. The wizard sets up the project and adds a blank form so that you can begin to add the controls to design the form.

The ActiveX Control wizard generates an implementation unit that descends from two objects, *TActiveXControl* for the ActiveX control specifics, and the VCL object of

the control that you choose to encapsulate. The ActiveForm wizard generates an implementation unit that descends from *TActiveXForm*.

To create a new ActiveX control, you must perform the following steps:

- 1 Design and create the custom VCL control that forms the basis of your ActiveX control.
- 2 Use the ActiveX control wizard to create an ActiveX control from a VCL control
or,
Use the ActiveForm wizard to create an ActiveX control based on a VCL form for Web deployment.
- 3 Use the ActiveX property page wizard to create one or more property pages for the control (optional).
- 4 Associate the property page with the ActiveX control (optional).
- 5 Register the control.
- 6 Test the control with all potential target applications.
- 7 Deploy an ActiveX control on the Web.

An ActiveX control consists of the VCL control from which it is built, as well as properties, methods, and events that are listed in the control's type library.

Elements of an ActiveX control

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, properties, methods, events, and one or more associated type libraries.

VCL control

An ActiveX control in Delphi is simply a VCL control that has been made accessible to applications and objects that support ActiveX controls. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

Note The controls available from the wizard list are controls derived from *TWinControl*. Some controls, such as *EditControl*, are registered as a *NonActiveX* control, therefore, they do not appear in the list.

Type library

A type library contains the type definitions for the control and is created automatically by the ActiveForm wizard. This type information, which provides more details than the interface, provides a way for controls to advertise its services to host applications. When you design your control, type library information is stored in a file with the TLB extension and a corresponding Pascal file containing the Pascal

translations. When you build the ActiveX control, the type library information is automatically compiled into the ActiveX control DLL as a resource.

Properties, methods, and events

The properties, events, and methods of the VCL control become those of the ActiveX control.

- A property is an attribute, such as a color or label.
- A method is a request to a control to perform some action.
- An event is a notification from a control to the container that something has happened.

Property page

The property page allows the user of a control to view and edit its 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 “Enabling simple data binding of ActiveX controls in the Delphi container” on page 48-12.

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 this, see Part IV, “Creating custom components.”

When designing ActiveX controls from either existing VCL controls or from new ActiveForms, keep in mind that you are implementing an ActiveX control that 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.

Whether you create an ActiveX control from a VCL control or ActiveForm depends on whether you already have an existing control that you simply want to encase in an ActiveX control wrapper. In this case, use the ActiveX control wizard as described in, “Generating an ActiveX control from a VCL control,” on page 48-4.

If you are creating a more complete ActiveX application, use the ActiveForm wizard as described in “Generating an ActiveX control based on a VCL form” on page 48-6. Typically, you use an ActiveForm to create and supply ActiveX applications for the Web.

The wizards implement all the necessary ActiveX interfaces required using the VCL objects, *TActiveXControl* or *TActiveForm*. You need only implement any additional interfaces you may have added to your control.

Once you have generated the control using either wizard, you can modify the control’s properties, methods, and events using the Type Library editor.

Generating an ActiveX control from a VCL control

You generate an ActiveX control from a VCL control by using the ActiveX Control wizard. The properties, methods, and events of the VCL control become the properties, methods, and events of the ActiveX control.

For information about creating VCL controls, see Part IV, “Creating custom components.”

The ActiveX control wizard actually puts an ActiveX class wrapper around a VCL control and builds the ActiveX control that contains the object. This ActiveX wrapper exposes the capabilities of the VCL control to other objects and servers.

Before using the ActiveX control wizard, you must select the VCL control from which to build the ActiveX control.

To bring up the ActiveX control wizard,

- 1 Choose File | New to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveX Control icon.

In the wizard, specify the following:

VCL ClassName	Choose the VCL control from the drop-down list. For example, to create an ActiveX control that allows client applications to use a <i>TButton</i> object, select <i>TButton</i> . For ActiveForms, the VCL class name option is dimmed because Active forms are always based on <i>TActiveForm</i> .
New Name	The wizard supplies a default name that clients will use to identify your ActiveX control. Change this name to provide a different OLE class name.
Implementation Unit	The wizard supplies a default name for the unit that contains code to implement the behavior of the ActiveX control. Accept the default or type in a new name.
Project Name	The wizard supplies a default name for the ActiveX library project for your ActiveX control, if no project is currently open. If you have an ActiveX library open, this option is disabled.
Threading Model	Choose the threading model to indicate how client applications can call your control’s interface. This is the threading model that you commit to implementing in the control. For more information on threading models, see “Choosing a threading model” on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Specify the following ActiveX control options:

- | | |
|------------------------------------|---|
| Make Control Licensed | Check this box to enable licensing for the ActiveX controls that you design and distribute (unless the control is distributed free of charge). The wizard generates both a design-time license for designers of the control, and a runtime license for users of the control. |
| Include Version Information | Check this box to include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Specify version information by choosing Project Options and selecting the Version Info page. Click the Help button on this page for information about each setting.

Note: Version information is required for registering a control in Visual Basic 4.0. |
| Include About Box | When this box is checked, an About box is included in the project. The user of the control can display the About box in a development environment. The About box is a separate form that you can modify. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button. |

The wizard generates the following files:

- 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 (with the TLB extension), which defines and implements the properties, methods, and events that the ActiveX control exposes for Automation. For more information about the type library, refer to Chapter 50, "Working with type libraries."
- An ActiveX implementation unit, which defines and implements the ActiveX control using the Delphi ActiveX framework (DAX).
- An About box form and unit (if Include About Box is checked).

Licensing ActiveX controls

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically, when the control is created at runtime.

To provide design-time licenses, the ActiveX wizard creates a key for the control that is stored in a file with the same name as the project and with the LIC extension. 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 will have a separate key entry in the LIC file.

To support the creation of runtime licenses, the license is generated by querying the control for the license (at design time), storing it, and then passing the license to the control when it is created in the context of an EXE. When the runtime license is passed to the control, the design time license is no longer required.

Runtime licenses for the Internet Explorer requires 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 IE, 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 IE 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, it renders the control on the page. If more than one LPK is included in a Web page, IE ignores all but the first.

For more information, search for Licensing ActiveX Controls on the Microsoft Web site.

Generating an ActiveX control based on a VCL form

The ActiveForm wizard generates an ActiveX control based on a VCL form which you design when the wizard leaves you in the Form Designer. You can use the ActiveForm to create applications that you can deploy on the Web.

When an ActiveForm is deployed, an HTML page is created to contain the reference to the ActiveForm and specify its location within the page. The ActiveForm is then displayed and run from within a Web browser. Inside the Web browser, the form behaves just like a stand-alone Delphi form. The form may contain any VCL or ActiveX components, including custom-built VCL controls.

To start the ActiveForm wizard,

- 1 Choose File | New to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveForm icon.

In the wizard, specify the following:

VCL ClassName	The VCL class name option is dimmed because Active forms are always based on <i>TActiveForm</i> .
New Name	The wizard supplies a default name that clients will use to identify your ActiveX control. Change this name to provide a different OLE class name.
Implementation Unit	The wizard supplies a default name for the unit that contains code to implement the behavior of the ActiveX control. Accept the default or type in a new name.
Project Name	The wizard supplies a default name for the ActiveX library project for your ActiveX control, if no project is currently open. If you have an ActiveX library open, this option is disabled.
Threading Model	Choose the threading model to indicate how client applications can call your control's interface. This is the threading model that you commit to implementing in the control. For more information on threading models, see "Choosing a threading model" on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Specify the following ActiveX control options:

Make Control Licensed	Check this box to enable licensing for the ActiveX controls that you design and distribute (unless the control is distributed free of charge). The wizard generates both a design-time license for designers of the control, and a runtime license for users of the control.
Include Version Information	Check this box to include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Specify version information by choosing Project Options and selecting the Version Info page. Click the Help button on this page for information about each setting. Note: Version information is required for registering a control in Visual Basic 4.0.
Include About Box	When this box is checked, an About box is included in the project. The user of the control can display the About box in a development environment. The About box is a separate form that you can modify. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button.

The wizard generates the following files:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A form.
- A type library (with the TLB extension), which defines and implements the properties, methods, and events that the ActiveX control exposes for Automation. For more information about the type library, refer to Chapter 50, "Working with type libraries."
- An ActiveX implementation unit, which defines and implements the properties, methods, and events of *TActiveForm* using the Delphi ActiveX framework (DAX).
- An About box form and unit (if Include About Box is checked).

At this point, the wizard adds a blank form to your ActiveX library project. Now 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 will create a test HTML page with a reference to the ActiveForm.

Working with properties, methods, and events in an ActiveX control

The ActiveX Control and ActiveForm wizards generate a type library that defines and implements the properties, methods, and events of the original VCL control or form, with the following exceptions:

- Any property, method, or event that uses a non-OLE type.
- Any property that is data-aware.

Therefore, you may need to add some properties, methods, and events manually.

Note Since ActiveX controls have a different mechanism for making controls data-aware than a VCL control, the wizards do not convert the properties related to data. You may want to add some of the ActiveX data aware properties to your control as described in "Enabling simple data binding with the type library" on page 48-11.

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library using either of the following ways:

- Choose Edit | Add To Interface from the IDE. For details, see "Adding additional properties, methods, and events" on page 48-9.
- Use the Type Library editor as described in Chapter 50, "Working with type libraries."

Note Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

Note Non-published properties that are manually added to the type library 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, the changes will not be reflected when the control is run. If the source is a VCL object and the property is not already published, you must create a descendant of the VCL object and publish the property in the descendant.

Adding additional properties, methods, and events

You can add additional properties, methods, and events to the control as follows.

- 1 Choose Edit | Add To Interface. The ActiveX controls implementation unit must be open and selected for the menu item to be available.

This brings up the Add to Interface dialog box.

- 2 Choose Method or Event from the Interface drop-down list. Next to Properties/ Methods or Events is the name of the interface to which the member will be added.

- 3 Enter the declaration for the property, method, or event. For example, if you were creating an ActiveX control with the *TButton* VCL control, you could add the following property:

```
property Top:integer;
```

If you check the Syntax helper check box, pop-up windows appear as you type, prompting you for what is expected at each point.

- 4 Choose OK.

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 actually depends on whether you have added a property, method, or event.

How Delphi adds properties

Since the interface is a dual interface, the properties are implemented using read and write access methods in the Pascal file (with the PAS extension). When adding a Caption property, the wizard would actually add the following declaration to the implementation unit:

```
property Caption: Integer read Get_Caption write Set_Caption;
```

The read and write access method declarations and implementations for the property would be the following:

```
function Get_Caption: Integer; safecall;
procedure Set_Caption(Value: Integer); safecall;

function TButtonX.Get_Caption: Integer;
begin
    Result := FDelphiControl.Caption;
end;
procedure TButtonX.Set_Caption(Value: Integer);
begin
    FDelphiControl.Caption := Value;
end;
```

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.

The interface implementation methods simply pass through the behavior to the VCL control. In some cases, you may need to add code to convert the COM data types to native Delphi types.

How Delphi adds methods

When you add a method, an empty implementation is added for you to add its functionality. For example, if you added:

```
procedure Move;
```

the following declaration and code would be added to the unit:

```
procedure Move; safecall;
procedure TButtonX.Move;
begin
end;
```

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

An ActiveX control can fire events to its container. You add events to specify which events can be fired by the control. The following items are created for each event on the control:

- A dispatch interface declaration and implementation entries in the implementation unit.

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);

procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;
```

- A type library entry for the event interface with dispinterface checked on its attributes page.
- An Object Pascal version of the type library's interface source file.

Unlike the Automation interface, the events interface is not added to the control's interface list. The control does *not* receive these events—the *container* does.

Enabling simple data binding with the type library

With simple data binding, you can bind a property of your ActiveX control to a specific field within a database. You can do so by setting the property's binding flags using the Type Library editor.

This section describes how to bind data-aware properties in an ActiveX control. For information on binding data-aware properties in a Delphi container, see "Enabling simple data binding of ActiveX controls in the Delphi container" on page 48-12.

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies the database that the value has changed and requests that the database record be updated. The database then notifies the control whether it succeeded or failed to update the record.

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 make a *TEdit* control 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.

Enabling simple data binding of ActiveX controls in the Delphi container

After installing a data-aware ActiveX control in the ActiveX tab of the Palette, and placing the control in the form designer, right-click the data-aware ActiveX control to display a list of options. In addition to the basic options, the additional DataBindings item appears.

Note: You must set the data source property to the data source component on the form before invoking the Data Bindings Editor. In doing so, the dialog supplies the Field Name and Property fields from the data source component. The editor lists only those properties from the data source component that can be data-bound properties of the ActiveX control.

To bind a field to a property,

1 In the ActiveX Data Bindings Editor dialog, select a field and a property name.

Field Name lists the fields of the database and Property Name lists the ActiveX control properties that can be bound to a database field. The DispID of the property is in parentheses, for example, Value(12).

2 Click Bind and OK.

Note: If no properties appear in the dialog, the ActiveX control contains no data-aware properties. To enable simple data binding for a property of an ActiveX control, use the type library as described in “Enabling simple data binding with the type library” on page 48-11.

The following example walks you through the steps of using a data-aware ActiveX control in the Delphi container. In this example, we use the Microsoft Calendar Control, which is available if you have Microsoft Office 97 installed on your system.

- 1 From the Delphi main menu, choose Component | Import ActiveX.
- 2 Select a data-aware ActiveX control, such as the Microsoft Calendar control 8.0, change its class name to *TCalendarAXControl*, and click Install.
- 3 In the Install dialog, click OK to add the control to the default user package, which makes the control available on the Palette.
- 4 Choose Close All and File | New Application to begin a new application.
- 5 From the ActiveX tab, drop a *TCalendarAXControl* object, which you just added to the Palette, onto the form.
- 6 From the Data Access tab, drop a DataSource and Table object onto the form.
- 7 Select the DataSource object and set its DataSet property to Table1.
- 8 Select the Table object and do the following:
 - Set the DataBaseName property to DBDEMOS
 - Set the TableName property to EMPLOYEE.DB
 - Set the Active property to True

- 9 Select the *TCalendarAXControl* object and set its *DataSource* property to *DataSource1*.
- 10 Select the *TCalendarAXControl* object, right-click, and choose *Data Bindings* to invoke the *ActiveX Control Data Bindings Editor*.
Field Name lists all the fields in the active database. Property Name lists those properties of the ActiveX Control that can be bound to a database field. The *DispID* of the property is in parentheses.
- 11 Select the *HireDate* field and the *Value* property name, choose *Bind*, and *OK*.
The field name and property are now bound.
- 12 From the *Data Controls* tab, drop a *DBGrid* object onto the form and set its *DataSource* property to *DataSource1*.
- 13 From the *Data Controls* tab, drop a *DBNavigator* object onto the form and set its *DataSource* property to *DataSource1*.
- 14 Run the application.
- 15 Test the application as follows:
With the *HireDate* field displayed in the *DBGrid* object, navigate through the database using the *Navigator* object. The dates in the ActiveX control change as you move through the database.

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 on 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. For details see “*Exposing properties of an ActiveX control*” on page 48-15.

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.
- 2 Select the ActiveX tab.
- 3 Double-click the Property Page icon.

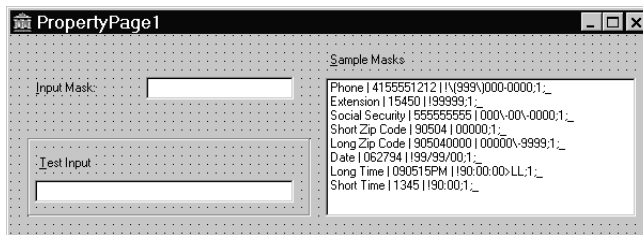
The wizard creates a new form and implementation unit for the property page.

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.

Figure 48.1 Mask Edit property page in design mode



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 you have added all 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 implementation unit. Specifically, you must add code to the *UpdatePropertyPage* and *UpdateObject* methods.

Updating the property page

Add code to the *UpdatePropertyPage* method to update the property's 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.

For example, the following code updates the property page's edit box control (InputMask) with the current value of the ActiveX control's (OleObject's) EditMask property:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
    { Update your controls from OleObjects }
    InputMask.Text := OleObject.EditMask;
end;
```

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.

For example, the following code sets the *EditMask* property of an ActiveX control (OleObject) using the value in the property page's edit box control (InputMask):

Note: Include the .TLB file that declares the *ICoClassNameDisp* interface.

```
procedure TPropertyPage1.UpdateObject;
begin
    {Update OleObjects from your controls }
    OleObject.EditMask := InputMask.Text;
end;
```

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 controls implementation for the unit. For example,

```
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
    DefinePropertyPage(Class_PropertyPage1);
end;
```

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; it is generated automatically by the Property Page wizard.

- 2 Add the property page unit to the **uses** clause of the controls implementation unit.

Exposing properties of an ActiveX control

The properties of an ActiveX control or ActiveForm can appear in a development environment such as Visual Basic, Delphi, or C++Builder. The user of the control or form can then visually manipulate the properties of the control. For properties to

appear, they must be defined in the control's type library. For property changes to persist, they must be published. You can add properties to the type library manually, or let Delphi add them automatically. Delphi automatically adds published properties to the type library for you.

Note Non-published properties that are manually added to the type library 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, the changes will not be reflected when the control is run.

To expose a property for a control that you created using the ActiveX control wizard, go to the source of the VCL control to expose the necessary properties. If the source is a VCL object and the property is not already published, you must create a descendant of the VCL object and publish the property in the descendant. For example, to publish the *Align* property of *TButton*, you can add the following code to the implementation unit of the ActiveX control:

```
TAlignButton = class (TButton)
published
  property Align;
end;
```

To expose a property for an ActiveForm,

- 1 Select Edit | Add to interface while the implementation unit is open in the editor.
- 2 Select Property/Method from the Procedure type drop-down list.
- 3 Enter the declaration for the property that will expose the control's property. For example, for the *Caption* property of a button control you could type,

```
property MyButtonCaption: WideString;
```

- 4 Choose OK.

This adds get and set methods to the implementation unit.

- 5 Add code to the methods to get and set the controls property. For example, with the button control above, the code would look as follows:

```
function TFormX.Get_MyButtonCaption: WideString;
begin
  Result := Button1.Caption;
end;
procedure TFormX.Set_MyButtonCaption(const Value: WideString);
begin
  Button1.Caption := Value;
end;
```

Since an ActiveForm is an ActiveX control, there is a type library associated with it, and you can add additional properties and methods to the ActiveForm. For information on how to do this, see "Working with properties, methods, and events in an ActiveX control" on page 48-8.

Registering an ActiveX control

After you have created your ActiveX control, you must register it so that other applications can find and use it.

To register an ActiveX control:

- Choose Run | Register ActiveX Server.

Note Before you remove an ActiveX control from your system, you should unregister it.

To unregister an ActiveX control:

- Choose Run | Unregister ActiveX Server.

As an alternative, you can use the **regsvr** command from the command line or run the **regsvr32.exe** from the operating system.

Testing an ActiveX control

To test your control, add it to a package and import it as an ActiveX control. This procedure adds the ActiveX control to the Delphi component palette. You can drop the control on a form and test as needed.

Your control should also be tested in all target applications that will use the control.

To debug the ActiveX control, select Run | Parameters and type the client name in the Host Application edit box.

The parameters then apply to the host application. Selecting Run | Run will run the host or client application and allow you to set breakpoints in the control.

Deploying an ActiveX control on the Web

Before the ActiveX controls that you create can be used by Web clients, they must be deployed on your Web server. Every time you make a change to the ActiveX control, you must recompile and redeploy it so that client applications can see the changes.

Before you can deploy your ActiveX control, you must have a Web Server that will respond to client messages. You can purchase a third-party Web Server, such as the Inprise Web Server that ships with the IntraBuilder product, or you can build your own Web Server if you have a version of Delphi that ships with socket components.

The process of deploying an ActiveX control,

- 1 Select Project | Web Deployment Options.
- 2 On the Project page, set the Target Dir to the location of the ActiveX control DLL as a path on the Web server. This can be a local path name or a UNC path, for example, C:\INETPUB\wwwroot.

- 3 Set the Target URL to the location as a Uniform Resource Locators (URL) of the ActiveX control DLL (without the file name) on your Web Server, for example, `http://mymachine.inprise.com/`. See the documentation for your Web Server for more information on how to do this.
- 4 Set the HTML Dir to the location (as a path) where the HTML file that contains a reference to the ActiveX control should be placed, for example, `C:\INETPUB\wwwroot`. This path can be a standard path name or a UNC path.
- 5 Set desired Web deployment options as described in “Setting options” on page 48-18.
- 6 Choose OK.
- 7 Choose Project | Web Deploy.

This creates a deployment code base that contains the ActiveX control in an ActiveX library (with the OCX extension). Depending on the options you specify, this deployment code base can also contain a cabinet (with the CAB extension) or information (with the INF extension).

The ActiveX library is placed in the Target Dir of step 2. The HTML file has the same name as the project file but with the HTM extension. It is created in the HTML Dir of step 4. The HTML file contains a URL reference to the ActiveX library at the location specified in step 3.

Note If you want to put these files on your Web server, use an external utility such as ftp.

- 8 Invoke your ActiveX-enabled Web browser and view the created HTML page.

When this HTML page is viewed in the Web browser, your form or control is displayed and runs as an embedded application within the browser. That is, the library runs in the same process as the browser application.

Setting options

Before deploying an ActiveX control, specify the Web deployment options that should be followed when creating the ActiveX library.

Web deployment options include settings to allow you to set the following:

Compress files by setting CAB file	A cabinet is a single file, usually with a CAB file extension, that stores compressed files in a file library. Cabinet compression can dramatically decrease download time (up to 70%) of a file. During installation, the browser decompresses the files stored in a cabinet and copies them to the user's system. Each file that you deploy can be CAB file compressed.
Sign the files	<p>Code signing allows the user of the control to determine with certainty who wrote the control and that the code has not been modified since it was signed. Each file that you deploy, including packages and additional files, can be code signed.</p> <p>Code signing generates a digital signature that is added to the file. ActiveX-enabled Web browsers that are set to enforce maximum security will not display ActiveX controls that are not code-signed.</p> <p>Note: Delphi does not provide the credentials and private key required for code signing. Information on how to obtain these files is available from Microsoft.</p>
Set options for packages	You can set specific options for each required package file as part of your deployment. These packages can each be code signed and put into CAB files. Packages that ship with Delphi are already code signed with the Inprise signature.

If you choose different combinations of package, CAB file compression, and code signing options for your ActiveX library, the resulting ActiveX library may be an OCX file, a CAB file containing an OCX file, or an INF file. See the Option combinations table below for details.

Web Deploy Options Default checkbox

Checking this box saves the current settings from the Project, Packages, Additional Files and Code Signing pages of the Web Deployment Options dialog box as the default options. If it is not checked, the settings affect only the open ActiveX project.

To restore the original settings, delete or rename the DEFPROJ.DOF file.

INF file

If your ActiveX control depends on any packages or other additional files, these must be included when you deploy the ActiveX control. When the ActiveX control is deployed with additional packages or other additional files, a file with the INF extension (for INFormation) is automatically created. This file specifies various files that need to be downloaded and set up for the ActiveX library to run. The syntax of the INF file allows URLs pointing to packages or additional files to download.

Web deployment options are displayed in the following tabs and described in sections below:

- Project tab
- Packages tab
- Additional files tab
- Code signing tab

Option combinations

The following table summarizes the results of choosing different combinations of package, CAB file compression and code signing Web deployment options.

Packages and/or additional files	CAB file compression	Code signing	Result
No	No	No	An ActiveX library (OCX) file.
No	No	Yes	An ActiveX library file.
No	Yes	No	A CAB file containing an ActiveX library file.
No	Yes	Yes	A CAB file containing an ActiveX library file.
Yes	No	No	An INF file, an ActiveX library file, and any additional files and packages.
Yes	No	Yes	A CAB file containing an INF file, an ActiveX library file, and any additional files and packages.
Yes	Yes	No	An INF file, a CAB file containing an ActiveX library, and a CAB file each for any additional files and packages.
Yes	Yes	Yes	A CAB file containing an INF file, a CAB file containing an ActiveX library, and a CAB file each for any additional files and packages.

Project tab

The project tab enables you to specify file locations, URL, and other deployment options for the project. The options on the project tab apply to the ActiveX library file or CAB file containing the ActiveX control and become the default options for any packages and additional files deployed with the project.

Directories and URLs	Meaning
Target Dir	Location as a path of the ActiveX library file on the Web server. This can be a standard path name or a UNC path. Example: C:\INETPUB\wwwroot
Target URL	Location as a Uniform Resource Locator (URL) of the path for the ActiveX library file on the Web Server. Example: http://mymachine.Inprise.com/
HTML Dir	Location as a path of the HTML file that contains a reference to the ActiveX library. This can be a standard path name or a UNC path. Example: C:\INETPUB\wwwroot

Note The locations specified are *paths* only, *not* complete file names.

In addition to specifying the locations for your ActiveX files, the project tab also allows you to specify whether to use CAB file compression, version numbers, code signing, and so on. The following table lists the options you can select:

General options	Meaning
Use CAB file compression	Compress the ActiveX library, and required packages and additional files, unless specified otherwise. Cabinet compression stores files in a file library, which can decrease a file's download time by up to 70%.
Include file version number	Include version number contained in Project Options VersionInfo.
Auto increment release number	Automatically increment the projects release number contained in Project Options VersionInfo.
Code sign project	Code sign the ActiveX library or CAB file containing the ActiveX library, and, unless specified otherwise, any DLLs or additional files to deploy. Code signing identifies the author and company of the control. Its digital signature assures users of the control that the control has not been modified since it was signed.
Deploy required packages	If checked, deploy the project's required packages.
Deploy additional files	If checked, deploy the additional files specified on the Addition Files tab with the project.

Packages tab

The packages tab lets you specify how to deploy the packages used by the project. Each package can have its own settings. When deploying your ActiveX control, you can specify individual deployment options for each required package file used by the project as part of your deployment. Each of these packages can be code signed and put into CAB files. Packages that ship with Delphi are already code signed with the Inprise signature.

Packages used by this project

To modify the settings for a particular package, select the package in the “Packages used by project” list box, and modify the settings as needed.

CAB options

CAB option	Meaning
Compress in a separate CAB	Create a separate CAB file for the package. This is the default.
Compress in a project CAB	Include the package in the project CAB file.

Output options

The output options allow you to specify if the package has version info and if it is code signed.

Output option	Meaning
Use file VersionInfo	Get the version info from the resource in the package file and enter it into the INF file for the project.
Code sign file	Code sign the package or CAB file containing the package.

Directory and URL options

Directory and URL option	Meaning
Target URL (or leave blank to assume file exists on target machines)	Location as a Uniform Resource Locators (URL) of the package on your Web Server. If left blank, the Web browser assumes the file exists on the target machine. If the package is not found on the target machine, the download of the ActiveX control fails.
Target directory (or leave blank to assume file exists on server)	Location as a path of the package on the Web server. This can be a standard path name or a UNC path. Leave blank to assume the file exists and ensure that it will not be overwritten.

Additional Files tab

The Additional Files tab enables you to specify additional files, such as .DLL files, .INI files, resources, etc., that must be deployed with the ActiveX control.

To add a file to be deployed, click on the Add button. This brings up a dialog from which you can browse and select a file. Each file you add is added to the “Files associated with project” list box.

Files associated with project

To modify the settings for a particular file, select the file in the “Files associated with project” list box, and modify the settings as needed.

CAB options

CAB option	Meaning
Compress in a separate CAB	Create a separate CAB file for the file. This is the default.
Compress in a project CAB	Include the file in the project CAB file.

Output options

Output option	Meaning
Use file VersionInfo	If the file contains VersionInfo resources, get the version info from the resource in the file and enter it into the INF file for the project.
Code sign file	Code sign the file or CAB file containing the file.

Directory and URL options

Directory and URL option	Meaning
Target URL (or leave blank to assume file exists on target machines)	Location as a Uniform Resource Locators (URL) of the file on your Web Server. If left blank, the Web browser assumes the file exists on the target machine. If the file is not found on the target machine, the download of the ActiveX library fails.
Target directory (or leave blank to assume file exists on server)	Location as a path of the file on the Web server. This can be a standard path name or a UNC path. Leave blank to assume the file exists and ensure that it will not be overwritten.

Code Signing tab

The Code Signing tab provides a way for users of the control to determine who wrote the control and to be assured that the control has not been modified since it was signed. Each file that you deploy can be code signed. Signing data does not alter it; it generates a digital signature that is added to the file.

To create a digital signature, a hash value (also known as a message digest) is first created using the selected Cryptographic digest algorithm. This hash value is then signed, using the Private key.

For the private key and credentials certificate files, contact Microsoft Corporation.

Required information

To code sign your ActiveX control, you must provide a private key and credentials certificate file. Specify these values in the Required Information section of the Code signing tab.

Required information	Meaning
Credentials file	To obtain this file, contact Microsoft.
Private key	The private key is known only to its owner and is used to generate the signature. To obtain this file, contact Microsoft.

Optional information

In addition to the required information, you can optionally add information to allow clients to obtain the application and company name:

Optional information	Meaning
Name of this application	Application name as you would like it to appear in the digital certificate displayed by the Web browser.
Application or company URL	A URL that provides a link to a Web tab for the product or company.

Timestamp server

Each vendor has a certificate that they use when code signing the file and this certificate must be renewed each year. Vendors often timestamp their digital signatures to verify that their vendor certificate was valid when the file was code signed. You can choose a timestamp server that generates the timestamp for your digital signature.

Timestamp server	Meaning
Default	Use the default timestamping server available on the Internet, Verisign.
None	Do not timestamp this control. Note that code signs without timestamps may become invalid.
Custom	Specify the name of a different timestamp server.
URL	Specify the location of the custom timestamp server on the Internet.

Cryptographic digest algorithm

Choose one of the following cryptographic digest algorithms. MD5 is the most commonly used and the default. You may want to select one algorithm over the other if your browser does not support the other.

Cryptographic digest algorithm	Meaning
MD5	The MD5 hashing algorithm was developed by RSA Data Security, Inc. This algorithm generates 128-bit hash value.
SHA 1	The SHA hashing algorithm was developed by the National Institute of Standards and Technology (NIST) and by the National Security Agency (NSA). This algorithm generates a 160-bit hash value.

Creating an Active Server Page

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 allow you to embed controls in a Web page that get called every time the server loads the Web page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and this control accesses data that gets updated every time the server loads the Web page.

Active Server Pages allow Web applications to be built using ActiveX server components (Automation objects). These are server-side components which you can develop using any Delphi or most other languages including C++, Java, Visual Basic. On the client side, the ASP is a standard HTML document and can be viewed by users on any platform using any Web browser.

Prior to this technology, client applications needed to be installed on every computer that would access the server. With this ASP model, most of the client application runs on the server; only the user interface presentation usually runs on the client. This makes it easier to update clients when an application changes.

You can also use Active Server Pages in conjunction with the Microsoft Transaction Server to automate the management of COM server components. For details on MTS, see Chapter 51, "Creating MTS objects."

This chapter shows how to create an Active Server Page using the Delphi Active Server Page wizard. With this, you can expose properties and methods of an existing application for Automation control.

Here are the steps for creating an Active Server Page object from an existing application:

- Create an Active Server Page object for the application.
- Expose the application's properties and methods for Automation.
- Register the application as an Active Server Page object.
- Test and debug the application.

For more background on the COM technologies, see Chapter 44, “Overview of COM technologies.” For information about creating an Automation controller, see Chapter 46, “Creating an Automation controller.”

Creating an Active Server Page object

An Active Server Page object is an Automation object that has access to the interfaces of a Web request. Like other Automation objects, it is an ObjectPascal class descending from *TAutoObject* that supports Automation protocols, exposing itself for other applications to use. You create an Active Server Page object using the Active Server Object wizard.

Before you create an Active Server Page 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.

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, you should be aware of the drawbacks of launching an out-of-process server.

To display the Active Server Object wizard:

- 1 Choose File | New.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Active Server Object icon.

In the wizard, specify the following:

CoClass Name	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)
Instancing	Specify an instancing mode to indicate how your Active Server Page object is launched. For details, see “COM object instancing types” on page 45-3. Note: When your Active Server Page object is used only as an in-process server, instancing is ignored.
Threading Model	Choose the threading model to indicate how client applications can call your object’s interface. This is the threading model that you commit to implementing in the Active Server Page object. For more information on threading models, see “Choosing a threading model” on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

- Active Server Type** Choose Page Level Event Methods with IIS 3 and IIS 4. It creates an Active Server Page object that implements the methods, *OnStartPage* and *OnEndPage*. These methods are called by the Web server when the page initializes and finishes execution. Use this with IIS 3 and IIS 4.
- Choose Object Context with IIS 5. It uses the functionality of MTS to retrieve the correct instance data of your object.
- Generate a template test script for this object** Generate a simple .ASP page which creates the Delphi object based off its ProgID.
- Generate event support code** Check this box to tell the wizard to implement a separate interface for managing events of your Active Server Page object. For more information on managing events, see "Managing events in your Automation object" on page 47-2.

When you complete this procedure, a new unit is added to the current project that contains the definition for the Active Server Page object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods of the interface through the type library as described in "Exposing an application's properties, methods, and events" on page 47-3.

The Active Server Page 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. For more information on dual interfaces, see "Dual interfaces" on page 47-6.

Creating ASPs for in-process or out-of-process servers

You can use `Server.CreateObject` in an ASP page to launch either an or server, depending on your requirements. However, launching in-process servers is more common.

In-process component DLLs are faster, more secure, and can be hosted by MTS, 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, you would receive 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 that run in-process with IIS or MTS. If performance and scalability are priorities for your site, we recommend that you do not use out-of-process components.

However, intranet sites that receive moderate to low traffic might be able to use an out-of-process component without adversely affecting the site's overall performance.

For general information on in-process and out-of-process servers, see, "In-process, out-of-process, and remote servers," on page 44-6.

Registering an application as an Active Server Page 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, it is recommended that you 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 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.

Testing and debugging the Active Server Page application

Debugging any in-process server such as an Active Server Page 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 Page 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 | 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 Page.
 - 5** Use the Web browser to interact with the Active Server Page.
- The Active Server Page pauses when the breakpoints are reached.

Working with type libraries

This chapter describes how to create and edit type libraries using Delphi's Type Library editor. Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by an ActiveX control or server. Type libraries provide a way for you to identify what types of objects and interfaces are available on your ActiveX server. For a detailed overview on why and when to use type libraries, see Chapter 44, "Overview of COM technologies."

By including a type library with your COM application or ActiveX library, you make information about the objects in your COM application available to other applications and programming tools.

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. With the Type Library editor, Delphi automates some of this process, easing the burden of creating your own type libraries.

If you create your COM object, ActiveX control, or Automation object using a wizard, the Type Library editor automatically generates the Pascal syntax for your existing object. Then you can easily refine your type library information using the Type Library editor. As you modify the type library using the Type Library editor, changes can be automatically updated in the associated object, or you can review and veto those changes, if the is enabled. (See page 50-34 for details.)

You can also use the Delphi Type Library Editor in the development of Common Object Request Broker Architecture (CORBA) applications. With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the Type Library editor and Delphi automatically updates the appropriate source files. For more information on CORBA, see Chapter 28, "Writing CORBA applications."

A type library can contain any and all of the following:

- Information about data types, including 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.

This chapter concludes with information on

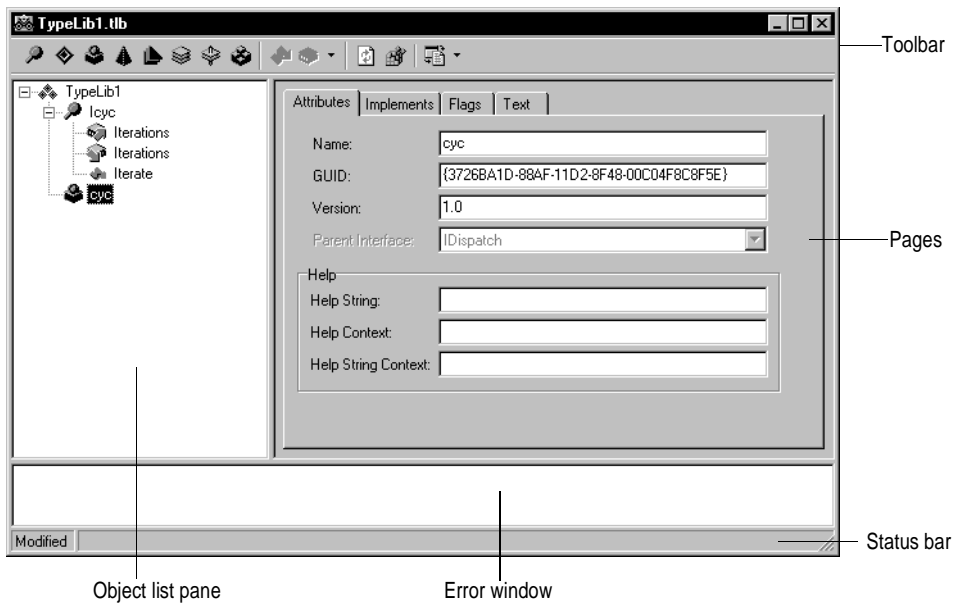
- Using Object Pascal or IDL syntax
- Creating new type libraries
- Deploying type libraries

Type Library editor

The Type Library editor is a tool that enables developers to examine and create type information for ActiveX controls and COM objects.

Figure 50.1 shows the Type Library editor displaying type information for an ActiveX button control.

Figure 50.1 Type Library editor



















The main elements of the Type Library editor are

- **Toolbar**, which you use to add new interfaces and interface members to your type library.
- **Object list pane**, which displays all the existing objects in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
- **Status bar**, which displays syntax errors if you try to add invalid types to your type library.
- **Pages**, which display information about the selected object. Which pages appear here depends on the type of object selected.
- **Error window**, which displays errors detected while loading a type library.

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.

You can add the following object types from the toolbar:

Icon	Meaning
	A type library. This can be expanded to expose the individual type information, including objects and interfaces.
	An interface description.
	A dispinterface description.
	A CoClass.
	An enumeration.
	An alias.
	A record.
	A union.
	A module.
	A method of the interface, dispinterface, or an entry point in a module.
	A put by value property function.
	A put by reference property function.
	A get property function.
	A property.
	A field in a record or union.
	A constant in an enum or a module.

The icons in the left box, Interface, Dispatch, CoClass, Enum, Alias, Record, Union, and Module represent the type info objects that you can edit.

When you click a toolbar button, the icon of that information type appears at the bottom of the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you selected, different pages of information appear to the right.

When you select an object, the Type Library editor displays which members are valid for that object. These members appear on the toolbar in the second box. For example, when you select Interface, the Type Library editor displays Method and Property

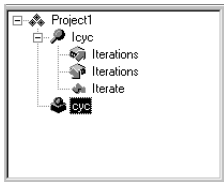
icons in the second box because you can add methods and properties to your interface definition. When you select Enum, the Type Library editor displays the Const member, which is the only valid member for Enum type information.

In the third box, you can choose to refresh, register, or export your type library, as described in “Saving and registering type library information” on page 50-33.

Object list pane

The Object list pane displays all the elements of the current type library, beginning with its interfaces. The interfaces expand to show the properties, methods, and events specified for that interface:

Figure 50.2 Object list pane



The context menu for the Object list pane provides the following options:

New	Pops up a menu containing a list of objects to add to your type library. These are the same objects available from the toolbar.
Cut	Removes the selected object and puts it on the Windows clipboard.
Copy	Puts the selected object on the Windows clipboard.
Paste	Inserts an object from the Windows clipboard below the selected object.
Delete	Removes the selected object.
View Errors	Toggles the visibility of the error window.
Toolbar	Toggles the visibility of the toolbar.

Status bar

When editing or saving a type library, syntax, translation errors, and warnings are listed in the Status bar 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” on page 50-23.

Pages of type information

When you select an object in the object list pane, pages of type information appear in the Type Library editor that are valid for the selected object. Which pages appear depends on the object selected in the object list panel as follows:

Table 50.1 Type library pages

Type Info object	Pages of type information
Type library	Attributes, Uses, Flags, Text
Interface	Attributes, Flags, Text
Dispinterface	Attributes, Flags, Text
CoClass	Attributes, Implements, Flags, Text
Enum	Attributes, Text
Alias	Attributes, Text
Record	Attributes, Text
Union	Attributes, Text
Module	Attributes, Text
Method	Attributes, Parameters, Flags, Text
Property	Attributes, Parameters, Flags, Text
Const	Attributes, Flags, Text
Field	Attributes, Flags, Text

Attributes page

All type library elements have attributes pages, which allow you to define a name and other attributes specific to the element. For example, if you have an interface selected, you can specify the GUID and parent interface. If you have a field selected, you can specify its type. Subsequent sections describe the attributes for each type library element in detail.

Attributes common to all elements in the type library are those associated with help. It is strongly recommended that you use the help strings to describe the elements in your type library to make it easier for applications using the type library.

The Type Library editor supports two mechanisms for supplying help. The traditional help mechanism, where a standard windows help file has been created for the library, or where the help information is located in a separate DLL (for localization purposes).

The following help attributes can apply to all elements:

Table 50.2 Attributes common to all types

Attribute	Meaning
Help String	A short description of the element.
Help Context	The Help context ID of the element, which identifies the Help topic within the standard windows Help file for this element. Used when a standard windows help file has been specified for the 'Help File' attribute of the type library.
Help String Context	The Help context ID of the element, which identifies the Help topic within the help DLL for this element. Used when a help DLL has been specified for the 'Help String DLL' attribute of the type library.

Note The help file must be supplied separately by the developer.

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 Object Pascal. 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 will generate 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 marshalling support).

Flags page

Some type library elements have flags that allow you to enable or disable certain characteristics or implied capabilities. Subsequent sections describe the flags for each type library element in detail.

Type library information

When the type library (top-most node) is selected in the Object list pane, you can change type information for the type library itself by modifying the following pages:

- Attributes page
- Uses page
- Flags page

Attributes page for a type library

The Attributes page displays type information about the currently selected type library:

Table 50.3 Type library attributes

Attribute	Meaning
Name	The descriptive name of the type library without spaces or punctuation.
GUID	The globally unique 128-bit identifier of the interface.
Version	A particular version of the library in cases where multiple versions of the library exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
LCID	The locale identifier that describes the single national language used for all text strings in the type library and elements.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.
Help String DLL	The fully qualified name of the DLL used for Help, if any.
Help File	The name of the help file associated with the type library, if any.

Uses page for a type library

The Uses page lists the names and locations of any other type libraries that this type library references.

Flags page for a type library

The following flags appear on the flags page when a type library is selected. It specifies how other applications must use the server associated with this type library:

Table 50.4 Type library flags

Flag	Meaning
Restricted	Prevents the library from being used by a macro programmer.
Control	Indicates that the library represents a control.
Hidden	Indicates that the library exists but should not be displayed in a user-oriented browser.

Interface pages

The 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, which is the default, a dispinterface is also implied and can be accessed through OLE automation.

You can modify a type library interface by

- Changing attributes
- Changing flags
- Adding, deleting, or modifying interface members

Attributes page for an interface

The Attributes page lists the following type information:

Table 50.5 Interface attributes

Attribute	Meaning
Name	The descriptive name of the interface.
GUID	The globally unique 128-bit identifier of the interface (optional).
Version	A particular version of the interface in cases where multiple versions of the interface exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Parent Interface	The name of the base interface for this interface. All COM interfaces must ultimately derive from <i>IUnknown</i> .
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Interface flags

The following flags are valid when an interface is selected in the main object pane.

Table 50.6 Interface flags

Flag	Meaning
Hidden	Indicates that the interface exists but should not be displayed in a user-oriented browser.
Dual	Identifies an interface that exposes properties and methods through both <i>IDispatch</i> and directly through a <i>v-table</i> . Default setting.
Ole Automation	Indicates that an interface can only use Automation-compatible types. This flag is not allowed on dispinterfaces since they are Automation compatible by definition.
Non-extensible	Indicates that the interface is not intended to be used as a base interface for another interface.

Interface members

The Type Library editor allows you to define new or modify existing

- Properties
- Methods

You can also change property and method parameters with the Parameters page.

The interface is more commonly used than the dispinterface to describe the properties and methods of an object.

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.

Interface methods

When you select the Method icon to add a new method, the Type Library editor displays the valid pages for a method: attributes, parameters, flags, and text.

Method attributes

The attributes for an interface method are as follows:

Table 50.7 Method attributes

Attribute	Meaning
Name	Name of the member.
ID	Dispatch ID.
Invoke kind	Indicates whether this method or property is a function. For methods, specify function.

Method parameters

You supply parameters for methods as described in “Property and method parameters page” on page 50-12.

Method flags

The flags for an interface method are as follows:

Table 50.8 Method flags

Flag	IDL Identifier	Meaning
Replaceable	replaceable	The object supports <i>IConnectionPointWithDefault</i> .
Restricted	restricted	Prevents the property or method from being used by a programmer.
Source	source	Indicates that the member returns an object or VARIANT that is a source of events.
Bindable	bindable	Indicates that the property supports data binding.
Hidden	hidden	Indicates that the property exists but should not be displayed in a user-oriented browser.
UI Default	uidefault	Indicates that the type information member is the default member for display in the user interface.

Interface properties

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.

Note ActiveX properties specified as Write By Reference means the property is passed as a pointer rather than by value. Some applications, such as Visual Basic, use the 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.

Property attributes

The attributes for an interface property are as follows:

Table 50.9 Property attributes

Attribute	Meaning
Name	Name of the member.
ID	Dispatch ID.
Type	Type of property; this can be any valid type as specified in “Valid types” on page 50-23.
Invoke kind	Indicates whether this property is a getter function, putter function, or putter by reference.

Property flags

The flags that you can set for an interface property are as follows:

Table 50.10 Property flags

Flag	IDL Identifier	Meaning
Replaceable	replaceable	The object supports <i>IConnectionPointWithDefault</i> .
Restricted	restricted	Prevents the property or method from being used by a programmer.
Source	source	Indicates that the member returns an object or VARIANT that is a source of events.
Bindable	bindable	Indicates that the property supports data binding.
Request Edit	requestedit	Indicates that the property supports the OnRequestEdit notification.
Display Bindable	displaybind	Indicates a property that should be displayed to the user as bindable.
Default Bindable	defaultbind	Indicates the single, bindable property that best represents the object. Properties that have the defaultbind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Hidden	hidden	Indicates that the property exists but should not be displayed in a user-oriented browser.
Default Collection Element	defaultcollelem	Allows for optimization of code in Visual Basic.
UI Default	uidefault	Indicates that the type information member is the default member for display in the user interface.
Non Browsable	nonbrowsable	Indicates that the property appears in an object browser that does not show property values, but does not appear in a properties browser that does show property values.
Immediate Bindable	immediatebind	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.

Property and method parameters page

The parameters page allows you to specify the parameters and return values for your functions.

For property functions, the property type is derived from either the return type or the last parameter. Changing the type on the parameters page affects the property type displayed on the attributes page. Likewise, changing the type displayed on the attributes page, affects the contents of the parameters page.

Note Changing one property function affects any related property functions. The Type Library editor assumes that property functions are related if they have the same name and Dispatch ID.

The parameters page differs, depending on whether you are working in IDL or Object Pascal.

When you add a parameter, if you are working in Object Pascal, the Type Library editor supplies the following:

- Modifier
- Name
- Type
- Default Value

If you are working in IDL, the Type Library editor supplies the following:

- Name
- Type
- Modifier

You can change the name by typing a new name. Change the type by choosing a new value from the drop-down list. The possible values for type are those supported by the Type Library editor as listed in “Valid types” on page 50-23.

If you are working in Object Pascal, change the modifier by choosing a new value from the drop-down list. The possible values are as follows:

Table 50.11 Parameter modifiers (Object Pascal Syntax)

Modifier	Meaning
blank (the default)	Input parameter. This can be a pointer, but the value it refers to is not returned. (Same as In in IDL)
none	No information is provided for marshaling parameter values. This modifier should only be used with dispinterfaces, which are not marshaled. (same as having no flags in IDL)
out	Output parameter. This is a reference value that receives the result. (Same as Out in IDL)
optionalout	Output parameter that is optional. This must be a Variant type, and all subsequent parameters must be optional. (Same as [Out, Optional] in IDL)
var	Input/Output parameter. Combination of blank and out. (Same as [In, Out] in IDL)
optionalvar	Input/Output parameter that is optional. Combination of optional and optionalout. (Same as [In, Out, Optional] in IDL)
optional	Input parameter that is optional. This must be a variant type, and all subsequent parameters must be optional. (Same as [In, Optional] in IDL)
retval	Receives the return value. Return values must be the last parameter listed (as enforced by the Type Library editor). Parameters with this value are not displayed in user-oriented browsers. This is only applicable if the function is declared without the safecall directive. (same as [Out, RetVal] in IDL)

Use the default column to specify default parameter values. When you add a default, the Type Library editor automatically adds the appropriate flags to the type library.

To change a parameter flag (when working in IDL), double-click the Modifier field to go to the Parameters flag dialog box. You can select from the following parameter flags:

Table 50.12 Parameter flags (IDL syntax)

Flag	Meaning
In	Input parameter. This can be a pointer, but the value it refers to is not returned.
Out	Output parameter. This must be a pointer to a member that will receive the result.
RetVal	Receives the return value. Return values must be an out attribute, and be the last parameter listed (as enforced by the Type Library editor). Parameters with this value are not displayed in user-oriented browsers.
LCID	Indicates that this parameter is a locale ID. Only one parameter can have this attribute, it must be flagged as [in] and its type must be long. This allows members in the VTable to receive an LCID at the time of invocation. Parameters with this value are not displayed in user-oriented browsers. By convention, LCID is the parameter preceding RetVal. Not allowed in dispinterfaces.
Optional	Specifies an optional parameter. Of course, all subsequent parameters must also be optional.
Has Default Value	This allows you to specify a default value for a typed optional parameter. Value must be a constant that can be stored in a VARIANT.
Default Value	If you choose Has Default Value, supply the default here. The value must be the same type as the optional parameter.

Note When working in IDL, default values are specified using flags rather than in a separate column. Also, local IDs are specified using a flag rather than using a parameter type specifier of TLCID.

You can use the Move Up and Move Down buttons to change the order of your parameters. However, the editor will not change the order if the move breaks a rule of the IDL language. For example, the Type Library editor enforces the rule that return values must always be the last parameter in the parameter list.

Dispatch type information

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 modify a dispatch interface (dispinterface) by

- Changing attributes
- Changing flags
- Adding, deleting, or modifying dispinterface members

Attributes page for dispatch

The following attributes apply to the dispinterface:

Table 50.13 Dispinterface attributes

Attribute	Meaning
Name	The name by which the dispinterface is known in the type library. It must be a unique name within the type library.
GUID	The globally unique 128-bit identifier of the dispatch (optional).
Version	A particular version of the dispinterface in cases where multiple versions of the dispinterface exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Dispatch flags page

The flags page for Dispatch is the same as for Interface. See “Interface flags” on page 50-10.

Dispatch members

For your dispinterface, you can define

- Methods
- Properties

How you add methods and properties for dispinterfaces is the same as how you add them for interfaces, as described in “Property and method parameters page” on page 50-12.

Notice that when you create a property for a dispinterface, you can no longer specify a function kind or parameter types. Dispinterface method and property flags are the same as those for an interface as described in “Interface methods” on page 50-10 and “Interface properties” on page 50-11.

CoClass pages

The CoClass describes a unique COM object which implements one or more interfaces and specifies which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events.

You can modify a CoClass definition in the type library by

- Changing attributes
- Changing the implements page
- Changing flags

Attributes page for a CoClass

The attributes page that apply for the CoClass are as follows:

Table 50.14 CoClass attributes

Attribute	Meaning
Name	The descriptive name of the CoClass.
GUID	The globally unique 128-bit identifier of the CoClass (optional).
Version	A particular version of the CoClass in cases where multiple versions of the CoClass exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

CoClass Implements page

The Implements page is used to specify what interfaces and dispinterfaces are implemented for the CoClass. For each interface, the Implements page specifies whether the following items are supported:

Table 50.15 CoClass Implements page options

Interface	Description
Interface	Name of an interface that the CoClass implements.
GUID	Identifies the GUID of the member interface of the object.
Source	Indicates that the member is a source of events.
Default	Indicates that the interface or dispinterface represents the default interface. This is the interface that is returned by default when an instance of the class is created. A CoClass can have two default members at most. One represents the primary interface or dispinterface, and the other represents an optional dispinterface that serves as an event source.
Restricted	Prevents the item from being used by a programmer. A member of an interface cannot have both restricted and default attributes.
VTable	Enables an object to have two different source interfaces.

The implements page's context menu contains menu items to toggle the above options. The 'Insert interface' menu option brings up a dialog where you can choose an interface to add to the CoClass. The list includes interfaces that are defined in the current type library and interfaces defined in any type libraries that the current type library references.

CoClass flags

The following flags are valid when a CoClass is selected in the main object pane.

Table 50.16 CoClass flags

Flag	Meaning
Hidden	Indicates that the interface exists but should not be displayed in a user-oriented browser.
Can Create	Instance can be created with <code>CoCreateInstance</code> .
Application Object	Identifies the CoClass as an application object, which is associated with a full EXE application, and indicates that the functions and properties of the CoClass are globally available in this type library.
Licensed	Indicates that the CoClass to which it applies is licensed, and must be instantiated using <code>IClassFactory2</code> .
Predefined	The client application should automatically create a single instance of the object that has this attribute.
Control	Identifies a CoClass as an ActiveX control, from which a container site will derive additional type libraries or CoClasses.
Aggregatable	Indicates that the members of the class can be aggregated.
Replaceable	The object supports <code>IConnectionPointWithDefault</code> .

Enumeration type information

You can add or modify an Enumeration definition in the type library by

- Changing attributes
- Adding, deleting, or modifying enum members

Attributes page for an enum

The attributes page that apply to an enum are as follows:

Table 50.17 Enum attributes

Attribute	Meaning
Name	The descriptive name of the enum.
GUID	The globally unique 128-bit identifier of the enum (optional).
Version	A particular version of the enum in cases where multiple versions of the enum exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

It is strongly recommended that you provide a help string 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.

```
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
```

Enumeration members

Enums are comprised of a list of constants, which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default.

To define constants for your enum, click the New Const button.

Alias type information

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.

You can create or modify an alias definition in the type library by

- Changing attributes

Attributes page for an alias

The attributes page for an alias contains the following

Table 50.18 Alias attributes

Attribute	Meaning
Name	The descriptive name by which the alias is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the alias is not uniquely specified in the system.
Version	A particular version of the alias in cases where multiple versions of the alias exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Type	Type that you want to alias.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Record type information

You can add or modify a record definition in the type library by

- Changing attributes
- Adding, deleting, or modifying record members

Attributes page for a record

The attributes page for a record contain the following

Table 50.19 Record attributes

Attribute	Meaning
Name	The descriptive name by which the record is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the record is not uniquely specified in the system.
Version	A particular version of the record in cases where multiple versions of the record exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Record members

A record is comprised of a list of structure members or fields. A field has a

- Name
- Type

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Records can be defined with an optional tag.

Union type information

A union is a record with only a variant part.

You can add or modify a union definition in the type library by

- Changing attributes
- Adding, deleting, or modifying union members

Attributes page for a union

The attributes page for a union contain the following

Table 50.20 Union attributes

Attribute	Meaning
Name	The descriptive name by which the union is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the union is not uniquely specified in the system.
Version	A particular version of the union in cases where multiple versions of the union exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Union members

A union, like a record, is comprised of a list of structure members or fields. A field has the following parts:

- Name
- Type

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Unions can be defined with an optional tag.

Module type information

The module defines a group of functions, typically a set of DLL entry points.

You can create or modify a module definition in the type library by

- Changing attributes
- Adding, deleting, or modifying module members

Note Delphi does not automatically generate any declarations or implementation related to the module. The specified DLL is created by the user as a separate project.

Attributes page for a module

The attributes page for a module contains the following

Table 50.21 Module attributes

Attribute	Meaning
Name	The descriptive name of the module.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the module is not uniquely specified in the system.
Version	A particular version of the module in cases where multiple versions of the module exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
DLL name	Name of associated DLL for which these entry points apply.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Module members

A module can be comprised of

- Methods
- Constants

Module methods

Module methods have the following attributes:

Table 50.22 Module method attributes

Attribute	Meaning
Name	The descriptive name of the module
DLL entry	Entry point into the associated DLL

You supply parameters for module methods in the same way you supply interface parameters, which are described in “Property and method parameters page” on page 50-12.

Module constants

To define constants for your module, you supply the following:

- Name
- Value
- Type

A module constant can be either a numeric or a string, depending on the attribute. Numeric input is usually an integer in decimal or hexadecimal format; it can also be a single char constant (such as `\0`). String input is delimited by double quotation marks (`""`), and cannot span multiple lines. The backslash is an escape character. The backslash character followed by another character, including another backslash, prevents the second character from being interpreted with any special meaning. For example, to put a backslash into text, use:

```
"Pathname: c:\\bin\\"
```

Creating new type libraries

The Type Library editor enables you to create type libraries for ActiveX controls, ActiveX servers and other COM objects.

The editor supports a subset of valid types and safe arrays in a type library as described below.

This section describes how to:

- Create a new type library
- Open an existing type library
- Add an interface to the type library
- Add properties and methods to the type library
- Add a CoClass to the type library
- Add an enumeration to the type library
- Saving and registering type library information

Valid types

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Object Pascal. 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.

Table 50.23 Valid types

Pascal 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	No*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	No*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed real
Largeuint	uint64	VT_UI8	No	8 byte unsigned real
SYSINT	int	VT_INT	No*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	No*	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

Note For valid types for CORBA development, see Chapter 28, “Writing CORBA applications.”

* Word, LongWord, SYSINT, and SYSUINT may be Automation-compatible with some applications.

Note 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 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 expressed in the Interface Definition Language (IDL) syntax in the generated type library (.TLB) file in binary form.

If the parameter type is preceded by 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 will automatically call the COM API to create, copy, and destroy *SafeArrays*.

In the Type Library editor, a *SafeArray* must specify its component type. For example, in the following code, the *SafeArray* specifies a component type of Integer:

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

The component type for a *SafeArray* must be an Automation-compatible type. In the Object Pascal type library translation unit, the component type is not necessary or allowed.

Using Object Pascal or IDL syntax

By default, the Text page of the Type Library editor displays your type information using an extension of Object Pascal syntax. You can work directly in IDL instead by changing the setting in the Environment Options dialog. Choose Tools | Environment Options, and specify IDL as the Editor language on the Type Library page of the dialog.

Note The choice of Object Pascal or IDL syntax also affects the choices available on the parameters attributes page.

Like Object Pascal 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

Object Pascal 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.

Table 50.24 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

Table 50.24 Attribute syntax (continued)

Attribute name	Example	Applies to
propputref	[propputref]	members except CoClass members
public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass members
replaceable	[replaceable]	anything except CoClass members and parameters
requestededit	[requestededit]	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 Object Pascal 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 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);
  [propget, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Dispatch interface syntax

The Object Pascal 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 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 Object Pascal 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 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 Object Pascal 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 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 Object Pascal 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 IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record syntax

The Object Pascal 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 IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    helpstring 'Task description']
typedef struct
{
    long ID;
    DATE StartDate;
    DATE EndDate;
    BSTR Owname;
    SAFEARRAY (int) Subtasks;
} Tasks;
```

Union syntax

The Object Pascal 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 IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    helpstring 'item description']
typedef union
{
    BSTR Name;
    long ID;
    double Value;
} MyUnion;
```

Module syntax

The Object Pascal 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 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 an ActiveX control, for example, if you want to define a type library for an ActiveX control that is not yet implemented.

To create a new type library,

- 1 Choose File | New to open the New Items dialog box.
- 2 Choose the ActiveX page which opens the New page.
- 3 Select the Type Library icon.
- 4 Choose OK.

The Type Library editor opens with a prompt to enter a name for the type library.

- 5 Enter a name for the type library.

Opening an existing type library

When you use the wizards to create an ActiveX control, Automation object, ActiveForm, COM object, MTS object, Remote Data Module, or MTS Data Module, a type library is automatically created with an implementation unit.

To open an existing type library independent of a project,

- 1 Choose File | Open to open the Open dialog box.
- 2 In File Type, choose type library extension for a list of available type libraries.
- 3 Select the desired type library.
- 4 Choose Open.

Or, to open a type library associated with the current project,

- 1 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 ActiveX control. If you would prefer to review the changes beforehand, 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 | Environment Options | Type Library page. For more information, see , “Apply Updates dialog,” on page 50-34.

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 in 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.

Adding properties and methods to an interface or dispinterface

To add members to an interface or dispinterface,

- 1 Select the interface, and choose either a property or method icon from the toolbar.

An interface 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 attributes in its attributes page that you can modify to suit the member.

You can add properties and methods by typing directly into the text page using the Pascal syntax. For example, you can type the following property declarations into the text page of an interface:

```
property AutoSelect: WordBool; dispid 1;
property AutoSize: WordBool; dispid 2;
property BorderStyle: BorderStyle; dispid 3;
```

After you have added members to an interface member page, the members appear as separate items in the object list pane each with its own attributes page where you can modify each new member’s attributes.

If you have the Apply Updates dialog enabled, the Type Library editor will notify you before updating the sources when you save the type library. and will warn you of potential problems. 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.

Adding a CoClass to the type library

To add a CoClass to a type library,

- 1 On the toolbar, click on the CoClass icon.

A CoClass is added to the object list pane prompting you to add a name.

- 2 Type a name for the class.

The new class contains default attributes in its attributes page that you can modify to suit the class. To add members,

- 3 Right-click in the text page for the class to display a list of interfaces from which you can choose.

The list includes interfaces that are defined in the current type library and defined in any type libraries that the current type library references.

- 4 Double-click on an interface you want the class to implement.

The interface is added to the page with with its GUID and other attributes.

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 prompting you to add a name.

- 2 Type a name for the member.

The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by clicking on the New Const button. Then, select each enumerated value and assign its attributes using the attributes page.

Saving and registering type library information

After modifying your type library, you'll want to save and register the type library information. If the type library was created with one of the ActiveX server project types or objects, saving the type library automatically updates the binary type library, the Object Pascal code representing its contents, and the implementation code maintained for it.

The Type Library editor stores type library information in two formats:

- As an OLE compound document file, called *project.TLB*.
- As a Delphi unit.

This unit is the compilation of the declarations of the elements that are defined in the type library in Object Pascal terms. Delphi uses this unit to bind the type library as a resource in your .OCX or .EXE file. Make changes to the type library within the Type Library editor, as the editor generates these files each time you save the Type Library.

Note The type library is stored as a separate binary (.TLB) file, but is also linked into the server (.EXE, DLL, or .OCX).

Note When using the Type Library editor for CORBA interfaces, this unit defines the stub and skeleton objects required by the CORBA application.

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 Delphi unit to disk.
- **Refresh**, to update the Delphi 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 source file of the associated object. Optionally, you can review the changes 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 | Environment Options | Type Library page (which is on 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 a Delphi unit.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with an ActiveForm, ActiveX control, or Automation object.

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 the unit to disk.
- Notifies the module manager to update the implementation if the type library is associated with an ActiveForm, ActiveX control, or Automation object.

To refresh the type library choose the Refresh icon on the Type Library editor toolbar.

Note If you have renamed or deleted items from 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.

Registering the type library

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 an IDL file that contains the type information declarations. This file can describe the type information in either CORBA IDL or Microsoft IDL.

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 ActiveX 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:

- 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
```

There can be multiple type library resources in an ActiveX library. Application developers use the resource compiler to add the .TLB file to their own ActiveX library.

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.

Creating MTS objects

MTS is a robust runtime environment that provides transaction services, security, and resource pooling for distributed COM applications.

Delphi provides an MTS Object wizard that creates an MTS object so that you can create server components that can take advantage of the benefits of the MTS environment. MTS provides many underlying services to make creating COM clients and servers, particularly remote servers, easier to implement.

MTS components provide a number of low-level services, such as

- Managing system resources, including processes, threads, and database connections so that your server application can handle many simultaneous users
- Automatically initiating and controlling transactions so that your application is reliable
- Creating, executing, and deleting server components when needed
- Providing role-based security so that only authorized users can access your application

By providing these underlying services, MTS allows you to concentrate on developing the specifics for your particular distributed application. With MTS, you implement your business logic into MTS objects, or in MTS remote data modules. Upon building the components into libraries (DLLs), the DLLs are installed in the MTS runtime environment.

With Delphi, MTS clients can be stand-alone applications or ActiveForms. Any COM server can run within the MTS runtime environment.

This chapter provides an overview of the Microsoft Transaction Server (MTS) technology and how you can use it to write applications based on MTS objects. Delphi also provides support for an MTS remote data module, which is described in Chapter 14, "Creating multi-tiered applications."

Microsoft Transaction Server components

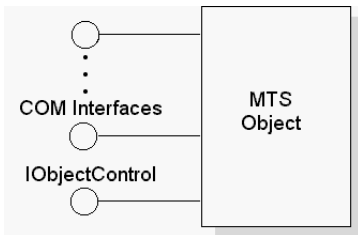
MTS components are COM in-process server components contained in dynamic-link libraries (DLLs). They are distinguished from other COM components in that they execute in the MTS runtime environment. You can create and implement these components with Delphi or any ActiveX-compatible development tool.

Note In MTS terms, a component represents the code that implements a COM object. For example, MTS components are implemented as classes in Delphi. MTS use of the term component interferes with Delphi traditional use of this term. We use component to refer to a class or object descending from the specific class, *TComponent*. However, to remain consistent with MTS terminology, we will use MTS component when talking specifically about MTS classes. In both MTS and Delphi, we use the term object to refer to an instance of an MTS component.

Typically, MTS server objects are small, and are used for discrete business functions. For example, MTS components can implement an application's business rules, providing views and transformations of the application state. Consider, for example, the case of a physician's medical application. Medical records stored in various databases represent the persistent state of the medical application, such as a patient's health history. MTS components update that state to reflect such changes as new patients, blood test results, and X-ray files.

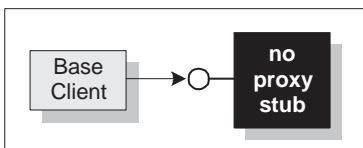
As shown in Figure 51.1, an MTS object can be viewed as any other COM object. In addition to supporting any number of COM interfaces, it also supports MTS interfaces. Just as *IUnknown* is the interface common to all COM objects, *IObjectControl* is common to all MTS objects. *IObjectControl* contains methods to activate and deactivate the MTS object and to handle resources such as database connections.

Figure 51.1 MTS object interface



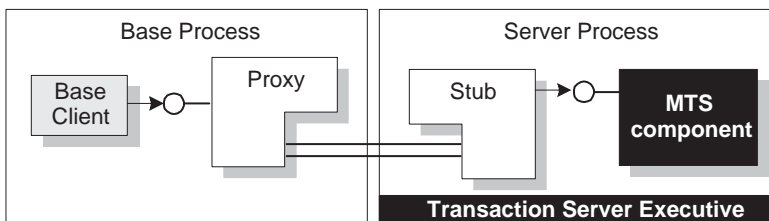
A client of a server within the MTS environment is called a **base client**. From a base client's perspective, a COM object within the MTS environment looks like any other COM object. The MTS object is installed as a DLL into the MTS executive. By running with the MTS EXE, MTS objects benefit from the MTS runtime environment features such as resource pooling and transaction support.

The MTS executive (.EXE) can be running in the same process as the base client as shown in Figure 51.2.

Figure 51.2 MTS In-process component

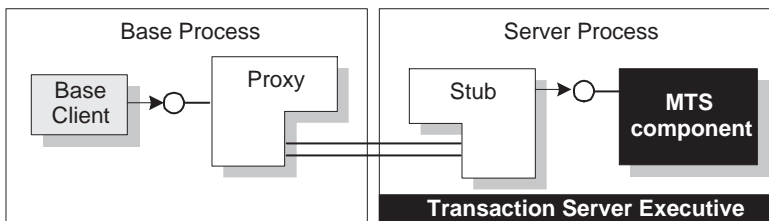
In-Process Application Component

The MTS component can be installed in a remote server process within the same machine as shown in Figure 51.3. The base client talks to a proxy which marshals the client's request to the MTS component's stub, which, in turn, accesses the MTS component through its interface.

Figure 51.3 An MTS component in an out-of-process server

Computer 1
In-Process Application Component

The MTS component can be installed in a remote server process on a separate computer as shown in Figure 51.4. Just as in any other remote server process, the client and remote server communicate across machines using DCOM.

Figure 51.4 An MTS component in a remote server process

Computer 1
Remote Component

Computer 2

Connection information is maintained in the MTS proxy. The connection between the MTS client and proxy remains open as long as the client requires a connection to the server, so it appears to the client that it has continued access to the server. In reality though, the MTS stub may deactivate and reactivate the object, conserving resources so that other clients may use the connection. For details on activating and deactivating, see "Managing resources with just-in-time activation and resource pooling" on page 51-4.

Requirements for an MTS component

In addition to the COM requirements, MTS requires that the component be a dynamic-link library (DLL). Components that are implemented as executable files (.EXE files) cannot execute in the MTS runtime environment.

In addition, an MTS component must meet the following requirements:

- When using the MTS Object wizard, the component must have a standard class factory, which is automatically supplied by Delphi.
- The component must expose its class object by exporting the standard *DllGetClassObject* method.
- All component interfaces and coclasses must be described by a type library, which is provided by the MTS Object wizard. You can add methods and properties to the type library by using the Type Library editor. The information in the type library is used by the MTS Explorer to extract information about the installed components during runtime.
- The component must only export interfaces that use standard COM marshaling, which is automatically supplied by the MTS Object wizard.
- Delphi's support of MTS does not allow manual marshaling for custom interfaces. All interfaces must be implemented as dual interfaces that use COM's automatic marshaling support.
- The component must export the *DllRegisterServer* function and perform self-registration of its CLSID, ProgID, interfaces, and type library in this routine. This is provided by the MTS Object wizard.
- A component running in the MTS process space cannot aggregate with components not running in MTS.

Managing resources with just-in-time activation and resource pooling

MTS manages resources by providing

- Just-in-time activation
- Resource pooling
- Object pooling

Just-in-time activation

The ability for an object to be deactivated and reactivated while clients hold references to it is called **just-in-time activation**. From the client's perspective, only a single instance of the object exists from the time the client creates it to the time it is finally released. Actually, it is possible that the object has been deactivated and reactivated many times. By having objects deactivated, clients can hold references to

the object for an extended time without affecting system resources. When an object becomes deactivated, MTS releases all the object's resources, for example, its database connection.

When a COM object is created as part of the MTS environment, a corresponding context object is also created. This context object exists for the entire lifetime of its MTS object, across one or more reactivation cycles. MTS uses the object context to keep track of the object during deactivation. This context object, accessed by the *IObjectContext* interface, coordinates transactions. A COM object is created in a deactivated state and becomes active upon receiving a client request.

An MTS object is deactivated when any of the following occurs:

- **The object requests deactivation with *SetComplete* or *SetAbort*:** An object calls the *IObjectContext SetComplete* method when it has successfully completed its work and it does not need to save the internal object state for the next call from the client. An object calls *SetAbort* to indicate that it cannot successfully complete its work and its object state does not need to be saved. That is, the object's state rolls back to the state prior to the transaction. Often, objects can be designed to be **stateless**, which means that objects deactivate upon return from every method.
- **A transaction is committed or aborted:** When an object's transaction is committed or aborted, the object is deactivated. Of these deactivated objects, the only ones that can continue to exist are the ones that have references from clients outside the transaction. Subsequent calls to these objects reactivate them and cause them to execute in the next transaction.
- **The last client releases the object:** Of course, when a client releases the object, the object is deactivated, and the object context is also released.

Resource pooling

Since MTS frees up idle system resources during a deactivation, the freed resources are available to other server objects. That is, a database connection that is no longer used by a server object can be reused by another client. This is called **resource pooling**.

Opening and closing connections to a database can be time-consuming. MTS uses resource dispensers to provide a way to reuse existing database connections rather than create new ones. A resource dispenser caches resources such as connections to a database, so that components within a package can share resources. For example, if you have a database lookup and a database update component running in a customer maintenance application, you would package those components together so that they can share database connections.

In the Delphi environment, the resource dispenser is the Borland Database Engine (BDE).

When developing MTS applications, you are responsible for releasing resources.

Releasing resources

You are responsible for releasing resources of an object. Typically, you do this by calling *SetComplete* and *SetAbort* methods after servicing each client request. These methods release the resources allocated by the MTS resource dispenser.

At this same time, you must release references to all other resources, including references to other objects (including MTS objects and context objects) and memory held by any instances of the component, such as using **free** in ObjectPascal.

The only time you would not include these calls is if you want to maintain state between client calls. For details, see “Stateful and stateless objects” on page 51-8.

Object pooling

Just as MTS is designed to pool resources, it is also designed to pool objects. After MTS calls the deactivate method it calls the *CanBePooled* method, which indicates that the object can be pooled for reuse. If *CanBePooled* is set to TRUE, rather than destroying the object upon deactivation, MTS moves the object to the object pool. Objects within the object pool are available for immediate use to any other client requesting this object. Only when the object pool is empty does MTS create a new object instance.

Objects that return FALSE or that do not support the *IObjectControl* interface are destroyed.

Note Object pooling and recycling is not available in this version of MTS. MTS calls *CanBePooled* as described, but no pooling takes place. This is provided for forward-compatibility to allow developers to use *CanBePooled* in their applications now so these applications can handle pooling when it becomes available. Currently, Delphi initializes *CanBePooled* to FALSE since object pooling is not yet available in MTS.

Accessing the object context

As with any COM object, a COM object using MTS must be created before it is used. COM clients create an object by calling the COM library function, *CoCreateInstance*.

Each COM object running in the MTS environment, must have a corresponding context object. This context object is implemented automatically by MTS and is used to manage the MTS component and coordinate transactions. The context object’s interface is *IObjectContext*. To access most methods of the object context, you can use the *ObjectContext* property of the *TMtsAutoObject* object. For example, you can use the *ObjectContext* property as follows:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the Object context is to use methods in the *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

You can use either of the above methods. However, there is a slight advantage of using the *TMtsAutoObject* methods rather than referencing the *ObjectContext* property when you are testing your application. For a discussion of the differences, see “Debugging and testing MTS objects” on page 51-20.

MTS transaction support

The transaction support provided by MTS allows you to group actions into transactions. For example, in a medical records application, if you had a Transfer component to transfer records from one physician to another, you could have your Add and Delete methods in the same transaction. That way, either the entire Transfer works or it can be rolled back to its previous state. Transactions simplify error recovery for applications that must access *multiple* databases.

MTS transactions ensure that

- All updates in a single transaction are either committed or get 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 and uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**. Resource managers use transaction-based synchronization protocols to isolate the uncommitted work of active transactions.
- Committed updates to managed resources (such as database records) survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**. Transactional logging allows you to recover the durable state after disk media failures.

When you declare that an MTS component is part of a transaction, MTS associates transactions with the component's objects. When an object's method is executed, the services that resource managers and resource dispensers perform on its behalf execute under a transaction. Work from multiple objects can be composed into a single transaction.

Transaction attributes

Every MTS component has a transaction attribute that is recorded in the MTS catalog. The MTS catalog maintains configuration information for components, packages, and roles. You administer the catalog using the MTS Explorer as described in “Administering MTS objects with the MTS Explorer” on page 51-22.

Each transaction attribute can be set to these settings:

Requires a transaction	MTS objects must execute <i>within the scope of a transaction</i> . When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction context, MTS automatically creates a new transaction context for the object.
Requires a new transaction	MTS objects must execute <i>within their own transactions</i> . When a new object is created, MTS automatically creates a new transaction for the object, regardless of whether its client has a transaction. An object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects.
Supports transactions	MTS objects can execute <i>within the scope of their client's transactions</i> . When a new object is created, its object context inherits the transaction from the context of the client. This enables multiple objects to be composed in a single transaction. If the client does not have a transaction, the new context is also created without one.
Does not support transactions	MTS objects <i>do not run within the scope of transactions</i> . When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction. Use this for COM objects designed prior to MTS support.

Object context holds transaction attribute

An object's associated context object indicates whether the object is executing within a transaction and, if so, the identity of the transaction.

Resource dispensers can use the context object to provide transaction-based services to the MTS object. For example, when an object executing within a transaction allocates a database connection by using the BDE resource dispenser, the connection is automatically enlisted on the transaction. All database updates using this connection become part of the transaction, and are either committed or aborted. For more information, see *Enlisting Resources in Transactions* in the MTS documentation.

Stateful and stateless objects

Like any COM object, MTS objects can maintain internal state across multiple interactions with a client. Such an object is said to be **stateful**. MTS objects can also be **stateless**, which means the object does not hold any intermediate state while waiting for the next call from a client.

When a transaction is committed or aborted, all objects that are involved in the transaction are deactivated, causing them to lose any state they acquired during the

course of the transaction. This helps ensure transaction isolation and database consistency; it also frees server resources for use in other transactions. Completing a transaction enables MTS to deactivate an object and reclaim its resources. See the following section for information on how to control when MTS releases your object state.

Maintaining state on an object requires the object to remain activated, holding potentially valuable resources such as database connections.

Note Stateless objects are more efficient and, therefore, they are recommended.

Enabling multiple objects to support transactions

You use *IObjectContext* methods as shown in the following table to enable an MTS object to participate in determining how a transaction completes. These methods, together with the component's transaction attribute, allow you to enlist one or more objects into a single transaction.

Table 51.1 IObjectContext methods for transaction support

Method	Description
SetComplete	Indicates that the object has successfully completed its work for the transaction. The object is deactivated upon return from the method that first entered the context. MTS reactivates the object on the next call that requires object execution.
SetAbort	Indicates that the object's work can never be committed. The object is deactivated upon return from the method that first entered the context. MTS reactivates the object on the next call that requires object execution.
EnableCommit	Indicates that the object's work is not necessarily done, but that its transactional updates can be committed in their current form. Use this to retain state across multiple calls from a client. When an object calls EnableCommit, it allows the transaction in which it is participating to be committed, but it maintains its internal state across calls from its clients until it calls SetComplete or SetAbort or until the transaction completes. EnableCommit is the default state when an object is activated. This is why an object should <i>always call SetComplete or SetAbort before returning from a method</i> , unless you want the object to maintain its internal state for the next call from a client.
DisableCommit	Indicates that the object's work is inconsistent and that it cannot complete its work until it receives further method invocations from the client. Call this before returning control to the client to maintain state across multiple client calls. This prevents the MTS runtime environment from deactivating the object and reclaiming its resources on return from a method call. Once an object has called DisableCommit, if a client attempts to commit the transaction before the object has called EnableCommit or SetComplete, the transaction will abort. You may use this, for example, to change the default state when an object is activated.

MTS or client-controlled transactions

Transactions can either be controlled directly by the client, or automatically by the MTS runtime environment.

Clients can have direct control over transactions by using a transaction context object (using the *ITransactionContext* interface). However, MTS is designed to simplify client development by taking care of transaction management automatically.

MTS components can be declared so that their objects always execute within a transaction, regardless of how the objects are created. This way, objects do not need to include any logic to handle the special case where an object is created by a client not using transactions. This feature also reduces the burden on client applications. Clients do not need to initiate a transaction simply because the component that they are using requires it.

With MTS transactions, you can implement the business logic of your application in your server objects. The server objects can enforce the rules so the client does not need to know about the rules. For example, in a physicians' medical application, an X-ray technician client can add and view X-rays in any medical record. It does not need to know that the application does not allow the X-ray technician to add or view any other type of medical record. That logic is in other server objects within the application.

Advantage of transactions

Allowing a component to either live within its own transaction or be part of a larger group of components that belong to a single transaction is a major advantage of the MTS runtime environment. It allows a component to be used in various ways, so that application developers can reuse application code in different applications without rewriting the application logic. In fact, developers can determine how components are used in transactions when packaging the component. They can change the transaction behavior simply by adding a component to a different package. For details about packaging components, see "Installing MTS objects into an MTS package" on page 51-21.

Transaction timeout

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging MTS objects.

To set the timeout value on your computer,

- 1 In the MTS Explorer, select Computer, My Computer.

By default, My Computer corresponds to the local computer on which MTS is installed.

- 2 Right-click and choose Properties and then choose the Options tab.

The Options tab is used to set the computer's transaction timeout property.

- 3 Change the timeout value to 0 to disable transaction timeouts.
- 4 Click OK to save the setting and return to the MTS Explorer.

For more information on debugging MTS applications, see “Debugging and testing MTS objects” on page 51-20.

Role-based security

MTS currently provides role-based security where you assign a role to a logical group of users. For example, a medical information application might define roles for Physician, X-ray technician, and Patient.

You define authorization for each component and component interface by assigning roles. For example, in the physicians' medical application, only the Physician may be authorized to view all medical records; the X-ray Technician may view only X-rays; and Patients may view only their own medical record.

Typically, you define roles during application development and assign roles for each package of components. These roles are then assigned to specific users when the application is deployed. Administrators can configure the roles using the MTS Explorer.

You can also set roles programmatically using the *ObjectContext* property *TMtsAutoObject*. For example,

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the object context is to use methods of the MTS *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

Note For applications that require stronger security, context objects implement the *ISecurityProperty* interface, whose methods allow retrieval of the Window's security identifier (SID) for the direct caller and creator of the object, as well as the SID for the clients which are using the object.

Resource dispensers

A resource dispenser manages the nondurable shared state on behalf of the application components within a process. Resource dispensers are similar to resource managers such as the SQL Server, but without the guarantee of durability. Delphi provides two resource dispensers:

- BDE resource dispenser
- Shared Property Manager

BDE resource dispenser

The BDE resource dispenser manages pools of database connections for MTS components that use the standard database interfaces, allocating connections to objects quickly and efficiently. For remote MTS data modules, connections are automatically enlisted on an object's transactions, and the resource dispenser can automatically reclaim and reuse connections.

Shared property manager

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. For example, you can use it to maintain the shared state for a multiuser game.

By using the Shared Property Manager, you avoid having to add a lot of code to your application; MTS provides the support for you. That is, the Shared Property Manager protects object state by implementing locks and semaphores to protect shared properties from simultaneous access. The Shared Property Manager eliminates name collisions by providing **shared property groups**, which establish unique name spaces for the shared properties they contain.

To use the Shared Property Manager resource, you first use the *CreateSharedPropertyGroup* helper function to create a shared property group. Then you can write all the properties to that group and read all the properties from that group. By using a shared property group, the state information is saved across all deactivations of an MTS object. In addition, state information can be shared among all MTS objects installed in the same package. You can install MTS components into a package as described in “Installing MTS objects into an MTS package” on page 51-21.

The following example shows how to add code to support the Shared Property Manager in an MTS object. After the example are tips for you to consider when designing your MTS application for sharing properties.

Example: Sharing properties among MTS object instances

The following example creates a property group called MyGroup to contain the properties to be shared among objects and object instances. In this example, we have a Counter property that is shared. It uses the *CreateSharedPropertyGroup* helper function to create the property group manager and property group, and then uses the *CreateProperty* method of the Group object to create a property called Counter.

To get the value of a property, you use the *PropertyByName* method of the Group object as shown below. You can also use the *PropertyByPosition* method.

```

unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private

```



```

    Group: ISharedPropertyGroup;
protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
    Exists: WordBool;
    Counter: ISharedProperty;
begin
    Group := CreateSharedPropertyGroup('MyGroup');
    Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
    Counter: ISharedProperty;
begin
    Counter := Group.PropertyByName['Counter'];
    Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
    Group := nil;
end;
initialization
    TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance, tmApartment);
end.

```

Tips for using the Shared Property Manager

For objects to share state, they all must be running in the same server process.

You can only use shared properties to share between objects running in the same process. If you want instances of different components to share properties, you must install the components in the same MTS package. Because there is a risk that administrators will move components from one package to another, it's safest to limit the use of a shared property group to instances of components that are defined in the same DLL.

Components sharing properties must have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same package.

Base clients and MTS components

It's important to understand the difference between clients and objects in the MTS runtime environment. Clients, or **base clients** in MTS terms, are not running under MTS. Base clients are the primary consumers of MTS objects. Typically, they provide the application's user interface or map the end-user's requests to the business functions defined in the MTS server objects. Alternatively, clients do not have the benefit of underlying MTS features. Clients do not get transaction support nor can they rely on resource dispensers.

The following table contrasts MTS components with base client applications.

Table 51.2 MTS server objects versus base clients

MTS components	Base clients
MTS components are contained in COM dynamic-link libraries (DLLs); MTS loads DLLs into processes on demand.	Base clients can be written as executable files (EXE) or dynamic-link libraries (DLL). MTS is not involved in their initiation or loading.
MTS manages server processes that host MTS components.	MTS does not manage base client processes.
MTS creates and manages the threads used by components.	MTS does not create or manage the threads used by base client applications.
Every MTS object has an associated context object. MTS automatically creates, manages, and releases context objects.	Base clients do not have implicit context objects. They can use transaction context objects, but they must explicitly create, manage, and release them.
MTS objects can use resource dispensers. Resource dispensers have access to the context object, allowing acquired resources to be automatically associated with the context.	Base clients cannot use resource dispensers.

MTS underlying technologies, COM and DCOM

MTS used the Component Object Model (COM) as its foundation to support the COM objects in client/server applications. COM defines a set of structured interfaces that enable components to communicate.

MTS uses DCOM for remote communication. To access a COM object on another machine, the client uses DCOM, which transparently transfers a local object request to the remote object running on a different machine. 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).

Resource pooling in the MTS environment is generally provided by an underlying database engine. In Delphi, resource pooling is provided by the Borland Database Engine. All database connections allocated by MTS objects come from this pool. These connections can participate in two-phase commit with MTS as the controller.

Overview of creating MTS objects

The process of creating an MTS component is as follows:

- 1 Use the MTS Object wizard to create an MTS component.
- 2 Add methods and properties to the application using the Type Library editor. For details on adding methods and properties using the Type Library editor, see Chapter 50, "Working with type libraries."
- 3 Debug and test the MTS component.
- 4 Install the MTS component into a new or existing MTS package.
- 5 Administer the MTS environment using the MTS Explorer.

Using the MTS Object wizard

Use the MTS Object wizard to create an MTS object that allows client applications to access your server within the MTS runtime environment. MTS provides extensive runtime support such as resource pooling, transaction processing, and role-based security.

To bring up the MTS Object wizard,

- 1 Choose File | New.
- 2 Select the tab labeled Multitier.
- 3 Double-click the MTS Object icon.

In the wizard, specify the following:

ClassName	Specify the name the for the MTS class.
Threading Model	Choose the threading model to indicate how client applications can call your object's interface. This is the threading model that you commit to implementing in the MTS object. For more information on threading models, see "Choosing a threading model" on page 45-3. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Transaction Model	Specify whether and how this MTS object supports transactions.
Generate event support code	Check this box to tell the wizard to implement a separate interface for managing events of your MTS object.

When you complete this procedure, a new unit is added to the current project that contains the definition for the MTS object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods

of the interface through the type library. You expose the interface as you would expose any Automation object as described in “Exposing an application’s properties, methods, and events” on page 47-3

The MTS object implements a **dual interface**, which supports both early (compile-time) binding through the vtable and late (runtime) binding through the *IDispatch* interface.

The MTS Object wizard implements the *IObjectControl* interface methods, Activate, Deactivate, and CanBePooled.

Choosing a threading model for an MTS object

The MTS runtime environment manages threads for you. MTS components should not create threads. Components must never terminate a thread that calls into a DLL.

When you specify the threading model when using the MTS wizard, you are specifying how the objects are assigned to threads for method execution.

Table 51.3 Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	No thread support. Client requests are serialized by the calling mechanism. All objects of a single-threaded component execute on the main thread. This is compatible with the default COM threading model, which is used for components that do not have a Threading Model Registry attribute or for COM components that are not reentrant. Method execution is serialized across all objects in the component and across all components in a process.	Allows components to use libraries that are not reentrant. Very limited scalability. Single-threaded, stateful components are prone to deadlocks. You can eliminate this problem by using stateless objects and calling SetComplete before returning from any method.
Apartment (or Single-threaded apartment)	Each object is assigned to a thread an apartment, which lasts for the life of the object; however, multiple threads can be used for multiple objects. This is a standard COM concurrency model. Each apartment is tied to a specific thread and has a Windows message pump.	Provides significant concurrency improvements over the single threading model. Two objects can execute concurrently as long as they are in different activities. These objects may be in the same component or in different components. Similar to a COM apartment, except that the objects can be distributed across multiple processes.

Note These threading models are similar to those defined by COM objects. However, because the MTS environment provides more underlying support for threads, the meaning of each threading model differs here. Also, the free threading model does not apply to objects running in the MTS environment due to the MTS support for activities.

MTS activities

MTS supports concurrency through **activities**. Every MTS object belongs to one activity, which is recorded in the object's context. The association between an object and an activity cannot be changed. An activity includes the MTS object created by the base client, as well as any MTS objects created by that object and its descendants. These objects can be distributed across one or more processes, executing on one or more computers.

For example, a physician's medical application may have an MTS object to add updates and remove records to various medical databases, each represented by a different object. This add object may use other objects as well, such as a receipt object to record the transaction. This results in several MTS objects that are either directly or indirectly under the control of the base client. These objects all belong to the same activity.

MTS tracks the flow of execution through each activity, preventing inadvertent parallelism from corrupting the application state. This feature results in a single logical thread of execution throughout a potentially distributed collection of objects. By having one logical thread, applications are significantly easier to write.

When an MTS object is created from an existing context, using either a transaction context object or an MTS object context, the new object becomes a member of the same activity. In other words, the new context inherits the activity identifier of the context used to create it.

MTS allows only a single logical thread of execution within an activity. This is similar in behavior to a COM apartment, except that the objects can be distributed across multiple processes. When a base client calls into an activity, all other requests for work in the activity (such as from another client thread) are blocked until after the initial thread of execution returns back to the client.

For more information on threading in the MTS environment, search the MTS documentation for the topic, Components and Threading.

Setting the transaction attribute

You set a transaction attribute either at design time or at runtime.

At design time, the MTS Object wizard prompts you to choose the transaction attribute.

You can change the transaction attribute at runtime by using the Type Library editor.

To change a transaction attribute at runtime,

- 1 Choose View | Type Library to open the Type Library editor.
- 2 Select the class corresponding to the MTS object.
- 3 Click the Transaction tab and choose the desired transaction attribute.

Note: If the MTS object is already installed into the runtime environment, you must first uninstall the object and reinstall it. Use `Run | Install MTS objects` to do so.

In addition, you can change the transaction attribute of an object installed in the MTS runtime environment by using the MTS Explorer.

Passing object references

You can pass object references, for example, for use as a callback, only in the following ways:

- Through return from an object creation interface, such as `CoCreateInstance` (or its equivalent), `ITransactionContext.CreateInstance`, or `IObjectContext.CreateInstance`.
- Through a call to `QueryInterface`.
- Through a method that has called `SafeRef` to obtain the object reference.

An object reference that is obtained in the above ways is called a **safe reference**. MTS ensures that methods invoked using safe references execute within the correct context.

Calls that use safe references always pass through the MTS runtime environment. This allows MTS to manage context switches and allows MTS objects to have lifetimes that are independent of client references.

Using the `SafeRef` method

An object can use the `SafeRef` function to obtain a reference to itself that is safe to pass outside its context.

The unit that defines the `SafeRef` function is `Mtx`.

`SafeRef` takes as input

- A reference to the interface ID (RIID) of the interface that the current object wants to pass to another object or client.
- A reference to the current object's `IUnknown` interface.

`SafeRef` returns a pointer to the interface specified in the RIID parameter that is safe to pass outside the current object's context. It returns `nil` if the object is requesting a safe reference on an object other than itself, or the interface requested in the RIID parameter is not implemented.

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call `SafeRef` first and then pass the reference returned by this call. An object should never pass a **self** pointer, or a self-reference obtained through an internal call to `QueryInterface`, to a client or to any other object. Once such a reference is passed outside the object's context, it is no longer a valid reference.

Calling `SafeRef` on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

When a client calls `QueryInterface` on a reference that is safe, MTS automatically ensures that the reference returned to the client is also a safe reference.

An object that obtains a safe reference must release the safe reference when it is finished with it.

For details on *SafeRef* see the *SafeRef* topic in the MTS documentation.

Callbacks

Objects can make callbacks to clients and to other MTS components. For example, you can have an object that creates another object. The creating object can pass a reference of itself to the created object; the created object can then use this reference to call the creating object.

If you choose to use callbacks, note the following restrictions:

- Calling back to the base client or another package requires access-level security on the client. Additionally, the client must be a DCOM server.
- Intervening firewalls may block calls back to the client.
- Work done on the callback executes in the environment of the object being called. It may be part of the same transaction, a different transaction, or no transaction.
- The creating object must call *SafeRef* and pass the returned reference to the created object in order to call back to itself.

Setting up a transaction object on the client side

A client base application can control transaction context through the *ITransactionContextEx* interface. The following code example shows how a client application uses *CreateTransactionContextEx* to create the transaction context. This method returns an interface to this object.

This example wraps the call to the transaction context in a call to `OleCheck` which is necessary because the `CreateInstance` method is not declared as **safecall**.

```

procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
    Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount: Currency);
var
    TransactionContextEx: ITransactionContextEx;
    CreditAccountIntf, DebitAccountIntf: IAccount;
begin
    TransactionContextEx := CreateTransactionContextEx;
    try
        OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
    
```

```
        CreditAccountIntf.Credit(Amount);  
    except  
        TransactionContextEx.Abort;  
        raise;  
    end;  
    TransactionContextEx.Commit;  
end;
```

Setting up a transaction object on the server side

To control transaction context from the MTS server side, you create an instance of *ObjectContext*. In the following example, the Transfer method is in the MTS object. In using *ObjectContext* this way, the instance of the object we are creating will inherit all the transaction attributes of the object who is creating it. We wrap the call in a call to *OleCheck* because the *CreateInstance* method is not declared as **safecall**.

```
procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;  
    Amount: Currency);  
var  
    CreditAccountIntf, DebitAccountIntf: IAccount;  
begin  
    try  
        OleCheck(ObjectContext.CreateInstance(DebitAccountId,  
            IAccount, DebitAccountIntf));  
        OleCheck(ObjectContext.CreateInstance(CreditAccountId,  
            IAccount, CreditAccountIntf));  
        DebitAccountIntf.Debit(Amount);  
        CreditAccountIntf.Credit(Amount);  
    except  
        DisableCommit;  
        raise;  
    end;  
    EnableCommit;  
end;
```

Debugging and testing MTS objects

You can debug local and remote MTS objects. When debugging MTS objects, you may want to turn off transaction timeouts.

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging MTS objects.

For information on remote debugging, see the Remote Debugging topic in Online help.

When testing the MTS object, you may first want to test your object outside the MTS environment to simplify your test environment.

While developing an MTS server, you cannot rebuild a server when it is still in memory. You may get a compiler error like, “Cannot write to DLL while executable is loaded.” To avoid this, you can set the package properties in the MTS Explorer to shut down the server when it is idle.

To shut down the MTS server when idle,

- 1 In the MTS Explorer, right-click the package in which your MTS component is installed and choose Properties.
- 2 Select the Advanced tab.
The Advanced tab determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.
- 3 Change the timeout value to 0, which shuts down the server as soon as no longer has a client to service.
- 4 Click OK to save the setting and return to the MTS Explorer.

Note When testing outside the MTS environment, you do not reference the *ObjectProperty* of *TMtsObject* directly. The *TMtsObject* implements methods such as *SetComplete* and *SetAbort* that are safe to call when the object context is *nil*.

Installing MTS objects into an MTS package

MTS applications consist of a group of in-process MTS objects (or MTS remote data modules) running in a single instance of the MTS executive (EXE). A group of COM objects that all run in the same process is called a **package**. A single machine can be running several different packages, where each package is running within a separate MTS EXE.

You can group your application components into a single package to run within a single process. You might want to distribute your components into different packages to partition your application across multiple processes or machines.

To install MTS objects into a package,

- 1 Choose Run | Install MTS Objects to install MTS objects into a package.
- 2 Check the MTS objects to be installed.
- 3 In the Install Object dialog box, choose the Into New Package to create a new package in which to install the MTS object or choose Into Existing Package to install the object into one of the existing MTS packages listed.
- 4 Choose OK to refresh the MTS catalog, which makes the objects available at runtime.

Packages can contain components from multiple DLLs, and components from a single DLL can be installed into different packages. However, a single component cannot be distributed among multiple packages.

Administering MTS objects with the MTS Explorer

Once you have installed MTS objects into an MTS runtime environment, you can administer these runtime objects using the MTS Explorer. The MTS Explorer is a graphical user interface for managing and deploying MTS components. With the MTS Explorer, you can

- Configure MTS objects, packages, and roles
- View properties of components in a package and view packages installed on a computer
- Monitor and manage transactions for MTS components that comprise transactions
- Move packages between computers
- Make a remote MTS object available to a local client

For details on the MTS Explorer, see the *MTS Administrator's Guide*.

Using MTS documentation

The documentation accompanying Microsoft's MTS provides thorough details of MTS concepts, programming scenarios, and administration tools. This documentation is likely to help those who are new to developing MTS applications.

Here is the roadmap of MTS documentation that accompanies the Microsoft product.

Table 51.4 Microsoft MTS documentation roadmap

Source	Description
<i>Setting Up MTS</i>	Describes how to set up MTS and MTS components, including instructions for accessing Oracle databases from MTS application and installing MTS sample applications.
<i>Getting Started with MTS</i>	Provides an overview of the new features in MTS, gives a brief tour of the documentation, and contains a glossary of terms.
<i>Quick Tour of MTS</i>	Provides an overview of MTS.
<i>MTS Administrator's Guide</i>	
Roadmap to the MTS Administrator's Guide	Describes the different ways to use the MTS Explorer to deploy and administer applications, and gives an overview of the MTS Explorer graphical interface.
Creating MTS Packages	Provides task-oriented documentation for creating and assembling MTS packages.
Distributing MTS Packages	Provides task-oriented documentation for distributing MTS packages.
Installing MTS Packages	Provides task-oriented documentation for installing and configuring MTS packages.
Maintaining MTS Packages	Provides task-oriented information for maintaining and monitoring MTS packages.
Managing MTS Transactions	Describes distributed transactions and the management of transactions using the MTS Explorer.

Table 51.4 Microsoft MTS documentation roadmap (continued)

Source	Description
Automating MTS Administration	Provides a conceptual overview, procedures, and sample code explaining how to use the MTS scriptable objects to automate procedures in the MTS Explorer.
<i>MTS Programmer's Guide</i>	
Overview and Concepts	Provides an overview of the product and how the product components work together, explains how MTS addresses the needs of client/server developers and system administrators, and provides in-depth coverage of programming concepts for MTS components.
Building Applications for MTS	Provides task-oriented information for developing ActiveX™ components for MTS.
MTS Administrative Reference	Provides a reference for using the MTS scriptable objects to automate procedures in the MTS Explorer.
MTS Reference	Provides a reference for the MTS application programming interface (API).

Index

Symbols

- & (ampersand) character 2-12, 5-19
- ... (ellipsis) buttons 26-21
- _ApplicationDisp, example 46-6

A

- Abort method 18-25
- AbortOnKeyViol property 20-23
- AbortOnProblem property 20-23
- About box 43-2, 43-3
 - ActiveForms 48-5, 48-7
 - adding properties 43-4
 - executing 43-5
- About unit 43-3
- AboutDlg unit 43-2
- abstract classes 31-3
- accelerators 2-12, 5-19
- access rights 20-4
- Access tables 13-8
 - isolation levels 13-8
 - local transactions 13-9
- access violations strings 3-29
- Acquire method 8-7
- action editor
 - adding actions 29-9
 - changing actions 29-10
- action items 29-7, 29-8, 29-10 to 29-13
 - adding 29-9
 - caution for changing 29-7
 - chaining 29-12
 - default 29-9, 29-11
 - enabling and disabling 29-11
 - event handlers 29-8
 - page producers and 29-19
 - responding to requests 29-12
 - selecting 29-10, 29-11
- action lists 2-25, 5-35 to 5-43
- Action property 5-37
- actions 5-35 to 5-43
 - centralizing 5-35, 5-36
 - clients 5-35
 - component writing 5-37
 - dataset 5-41
 - demos 5-43
 - executing 5-37
 - overview 5-35
 - predefined 5-39
 - registering 5-43
 - standard edit 5-39
 - standard window 5-40
 - targets 5-35
 - Update method 5-39
 - updating 5-39
 - using 5-36
- Actions property 29-9
- Active Data Objects 12-1
- Active Documents 44-9, 44-15
- Active property 23-23
 - ADO connection components 23-4
 - client sockets 30-6
 - datasets 18-3, 18-5
 - queries 21-12
 - server sockets 30-7
 - sessions 16-4
 - tables 20-4
- active record 18-9
 - canceling cached updates 25-8
 - setting 20-7
 - synchronizing 20-24
- Active Server Pages
 - creating 49-1 to 49-5
 - generate event support code 49-3
 - registering 49-4
 - testing and debugging 49-4
 - using the wizard 49-2
- Active Server Type 49-3
- ActiveFlag property 27-19, 27-20
- ActiveForms 48-6
 - as MIDAS Web applications 14-29
 - licenses 48-5, 48-7
 - wizards 48-6
- ActiveX 44-1, 44-11 to 44-12, 48-1
 - Web applications 44-11
- ActiveX controls 11-3, 44-9, 44-11, 44-18
 - associated files 48-22
 - CAB file compression 48-21
 - code signing 48-19
 - creating 48-1, 48-4 to 48-16
 - data-aware 48-8
 - elements 48-2
 - embedding in HTML 29-18
 - event handling 48-9, 48-10
 - from VCL controls 48-3
 - from VCL forms 48-6
 - licensing 48-5, 48-7
 - methods 48-9, 48-10
 - multi-tiered applications and 14-29
 - options 48-5
 - persistent properties 48-13
 - properties 48-3, 48-9
 - binding 48-11
 - publishing 48-16
 - property pages 48-12, 48-15
 - registering 45-6, 48-17
 - testing 45-6, 48-17
 - timestamps 48-24
 - type information 50-2
 - type libraries 44-13, 48-2, 50-36
 - version information 48-21
 - Web applications 44-11
 - Web deployment 48-17 to 48-25
 - wizard 48-1, 48-4
- ActiveX page (Component palette) 2-10
- ActiveX servers
 - optimizing 44-14
 - type checking 44-14
 - type libraries 44-12
- ActnList unit 5-36, 5-43
- Add Fields dialog box 19-6
- Add method
 - menus 5-26
 - persistent columns 26-19
 - queries 21-7
 - strings 2-31
- Add to Interface
 - command 14-16, 28-6
- Add To Repository
 - command 2-35
- AddAlias method 16-11
- AddFontResource
 - function 11-10
- Additional Files page (Web deployment) 48-22
- Additional page (Component palette) 2-10

- AddObject method 2-32
- AddPassword method 16-13
- AddRef method 3-16, 3-20, 3-21
 - Unknown 44-4
- Address property
 - client sockets 30-6
 - TSocketConnection 14-20
- addresses
 - socket connections 30-3, 30-4
- AddStandardAlias
 - method 16-11
- AddStrings method 2-31, 2-32
- ADO
 - creating tables 13-13
 - restructuring tables 13-13
 - SQL 23-26
- ADO 2.1 13-12
- ADO commands
 - canceling commands 23-27
 - executing commands 13-13, 23-27
 - parameters 23-17, 23-28
 - retrieving data 23-28
- ADO components 23-1 to 23-29
 - command components 23-26
 - connection components 23-3
 - data-aware controls 23-13
 - dataset components 23-11, 23-18
 - general descriptions 23-2
 - overview 23-1
 - query components 23-20
 - recordsets 23-14
 - stored procedure
 - components 23-22
 - table components 23-19
- ADO connections 23-2 to 23-11
 - activating 23-4
 - connecting to data
 - stores 23-2
 - connecting, overview 13-12
 - connection logins 23-7
 - fine-tuning a
 - connection 23-5, 23-6
 - overview 23-3
 - stored procedures,
 - listing 23-9
 - transactions 23-11
- ADO datasets 12-14, 23-1, 23-11 to 23-26
 - common dataset
 - features 23-12
 - connecting 23-3
 - connecting to data
 - stores 23-13
 - data files 23-14
 - executing commands 23-29
 - modifying data 23-12
 - navigating records 23-12
 - overview 23-18
 - retrieving data 13-12, 23-19
 - understanding 13-11
- ADO objects
 - Connection object 23-4
 - Field object 23-1
 - Properties object 23-1
- ADO page (Component palette) 12-1
- ADO queries 12-14
 - overview 23-20
 - SQL 13-13, 23-21
 - using 23-20
- ADO stored procedures 12-14
 - executing 23-23
 - overview 23-22
- ADO tables 12-14
 - overview 23-19
 - using 23-19
- ADO-based applications 13-10
- ADO-based architecture 13-11
- ADT fields 19-23, 19-24 to 19-25
- ADTG files 23-16
- AfterApplyUpdates event 15-6
- AfterClose event 18-5
- AfterConnect event 14-23
- AfterDisconnect event 14-23
- AfterDispatch event 29-10, 29-13
- AfterGetRecords event 15-5
- agentaddr file 28-19
- aggregate fields 19-6, 24-12
- aggregation
 - client datasets 24-9 to 24-12
 - interfaces 3-19
- aliases
 - database connections 16-6
 - database engine 16-9 to 16-11
 - database names and 13-3
 - deleting 16-11
 - remote connections 17-8
 - sessions and 13-4
 - specifying 17-4, 17-5
 - Type Library editor 50-19, 50-29
- AliasName property 17-4
- Align property 5-3
 - panels 5-29
 - status bars 2-20
 - text controls 6-6
- Alignment property 2-14, 19-2
 - column headers 26-22
 - data grids 26-21
 - decision grids 27-12
 - field values 19-12
 - memo fields 26-9
 - memos 2-13
 - rich text controls 2-13
 - status bars 2-20
- AllowAllUp property 2-16
 - speed buttons 5-30
 - tool buttons 5-32
- AllowDelete property 26-28
- AllowGrayed property 2-16
- AllowInsert property 26-28
- alternative indexes 20-8, 20-9
- alTop constant 5-29
- ampersand (&) character 2-12, 5-19
- analog video 7-31
- ancestor classes 2-2, 2-6, 32-3
 - default 32-3
- animation controls 2-22, 7-28 to 7-29
 - example 7-29
- ANSI character sets 10-2
- ANSI standard
 - strings 3-23
- AnsiChar 3-23
- AnsiString 3-25
- apartment threading 45-5
- Append method 18-7, 18-22
 - Insert vs. 18-21
- appending records
 - batch operations 20-19, 20-21
- AppendRecord method 18-24
- application files 11-2
- application servers 12-9, 14-1, 14-11 to 14-18
 - data providers 14-16, 15-1
 - identifying 14-19
 - remote data modules 2-34
- Application variable 5-3, 29-7
- applications
 - CGI 4-10
 - client/server 14-1, 17-1
 - network protocols 17-7
 - COM 4-11, 44-2, 44-16
 - combined COM and CORBA 28-4
 - CORBA 4-11, 28-1
 - database 12-1
 - deploying 11-1
 - distributed 4-9, 4-9 to 4-12
 - threads and 8-11

- graphical 31-7, 36-1
 - international 10-1
 - ISAPI 4-10, 29-5, 29-6
 - MDI 4-2
 - MIDAS Web
 - applications 14-28 to 14-38
 - MTS 4-11
 - multi-threaded 8-1, 16-2, 16-16 to 16-17
 - distributed 8-11
 - multi-tiered 14-1 to 14-38
 - overview 14-3 to 14-10
 - NSAPI 4-10, 29-5, 29-6
 - optimizing searches 20-5
 - realizing palettes 36-5
 - SDI 4-2
 - service 4-3
 - status information 2-20
 - synchronizing tables 20-24
 - undoing changes 18-23
 - Web server 4-10
 - Win-CGI 4-11
 - Apply method
 - update objects 25-18
 - Apply Updates dialog 50-34
 - ApplyRange method 20-14
 - ApplyUpdates method
 - cached updates 25-5, 25-6
 - client datasets 24-20
 - multi-tiered
 - applications 24-20
 - sockets 18-29
 - AppServer property 14-7, 14-17, 14-24
 - Arc method 7-4
 - architecture
 - BDE-based applications 13-2
 - client applications 14-4
 - CORBA applications 28-2 to 28-4
 - database applications 12-6
 - multi-tiered 14-4
 - Web-based 14-28
 - one-tiered 13-2
 - server applications 14-4
 - two-tiered 13-2
 - Web server applications 29-7
 - array fields 19-25 to 19-26
 - arrays 33-2, 33-8
 - safe 50-25
 - as reserved word
 - early binding 14-24
 - AS_ApplyUpdates method 14-8
 - AS_DataRequest method 14-8
 - AS_Execute method 14-8
 - AS_GetParams method 14-8
 - AS_GetProviderNames
 - method 14-8
 - AS_GetRecords method 14-8
 - AS_RowRequest method 14-8
 - AsBoolean function 19-19
 - ASCII tables 16-11, 20-3
 - AsCurrency function 19-19
 - AsDateTime function 19-19
 - AsFloat function 19-19
 - AsInteger function 19-19
 - Assign Local Data
 - command 13-16, 24-12
 - Assign method
 - string lists 2-32
 - AssignedValues property 26-18
 - assignment statements 33-2
 - object variables 2-7
 - AssignValue method 19-17
 - Associate property 2-14
 - as-soon-as-possible
 - deactivation 14-6
 - AsString function 19-19
 - AsVariant function 19-19
 - atomicity
 - transactions 51-7
 - attaching to databases 4-9
 - attribute specifications
 - type libraries 50-25
 - attributes
 - property editors 38-10
 - Attributes property 23-6
 - audio clips 7-30
 - AutoCalcFields property 18-26
 - AutoComplete property 14-6
 - AutoConnect 46-3
 - AutoDisplay property 26-10
 - graphics 26-11
 - rich edit controls 26-10
 - AutoEdit property 26-3, 26-7
 - AutoHotKeys property 5-19
 - Automation
 - creating objects 44-16
 - early binding 44-14
 - interfaces 47-6 to 47-8
 - late binding 47-8
 - optimizing 44-14
 - properties 47-7
 - type checking 47-6
 - type compatibility 47-9, 50-23
 - type descriptions 44-11, 47-7
 - Automation controller,
 - connecting to a server 46-3
 - Automation controllers 44-9, 44-10, 46-1 to 46-7, 47-7
 - accessing properties and methods 47-7
 - COM clients 44-8
 - creating from type libraries 46-1
 - example 46-4
 - IDispatch interface 47-7
 - using a dispatch interface 46-4
 - using a vtable interface 46-3
 - Automation objects 44-11
 - creating 47-1, 49-2
 - Automation server, connecting to 46-3
 - Automation servers 44-9, 44-10, 44-17
 - accessing objects 47-7
 - creating 47-1 to 47-8
 - testing 47-6, 49-4
 - AutoPopup property 5-34
 - AutoSelect property 2-13
 - AutoSessionName
 - property 16-17, 29-22
 - AutoSize property 5-4, 11-9, 26-9
 - .AVI files 7-31
 - AVI clips 2-22, 7-28, 7-31
 - axes values 27-15
- ## B
-
- backgrounds 10-8
 - Bands property 2-16, 5-33
 - base clients 51-2, 51-14
 - Basic Object Adaptor (BOA) 28-2, 28-14
 - batAppend constant 20-19, 20-21
 - batAppendUpdate
 - constant 20-19, 20-21
 - batch operations 20-18
 - appending data 20-21
 - copying datasets 20-21
 - deleting records 20-22
 - error handling 20-23
 - import modes 20-19
 - mapping data types 20-22
 - modes 20-21
 - running 20-23
 - setting up 20-20
 - updating data 20-21
 - BatchMove method 20-18
 - batCopy constant 20-19, 20-21
 - caution 20-19

- batDelete constant 20-19, 20-21
- batUpdate constant 20-19, 20-21
- BDE32.HLP 21-3
- BeforeApplyUpdates event 15-6
- BeforeClose event 18-5, 18-23
- BeforeConnect event 14-23
- BeforeDisconnect event 14-23
- BeforeDispatch event 29-9, 29-11
- BeforeGetRecords event 14-27, 15-5
- BeforeUpdateRecord event 15-9
- BeginDrag method 6-1
- BeginRead method 8-7
- BeginTrans method 23-11
- BeginWrite method 8-7
- Beveled 2-15
- beveled panels 2-22
- bevels 2-22
- bi-directional applications
 - methods 10-6 to 10-7
 - properties 10-5
- bi-directional cursors 21-15
- bitmap buttons 2-15
- bitmap frames 2-22
- bitmap object 7-3
- bitmaps 2-22, 36-4
 - adding scrollable 7-16
 - adding to components 38-3
 - associating with strings 2-32, 6-12
 - blank 7-17
 - brushes 7-8
 - brushes property 7-7, 7-8
 - destroying 7-20
 - drawing on 7-17
 - drawing surfaces 36-3
 - draw-item events 6-15
 - freeing memory 7-20
 - graphical controls vs. 40-3
 - in frames 5-14
 - internationalizing 10-9
 - loading 36-4
 - offscreen 36-5 to 36-7
 - replacing 7-19
 - ScanLine property 7-9
 - scrolling 7-16
 - setting initial size 7-16
 - temporary 7-16, 7-17
 - toolbars 5-31
 - when they appear in application 7-1
- BLOB fields 26-2
 - displaying values 26-9, 26-10
- fetch on demand 15-3, 24-17
- getting values 18-30
- updating 25-4
- viewing graphics 26-10
- BLOBs 26-9, 26-10
 - caching 18-30
- blocking connections 30-11 to 30-14
 - event handling 30-10
 - non-blocking vs. 30-10
- BMPDlg unit 7-20
- BOA 28-2, 28-14
- Bof property 18-10, 18-12
- Bookmark property 18-13
- bookmarks 18-12 to 18-14
- BookmarkValid method 18-13
- Boolean fields 26-2, 26-14
- Boolean values 33-2, 33-9, 42-3
- borders 2-11
 - panels 2-19
- BorderStyle property 2-11
- BorderWidth property 2-19
- Borland Database Engine 4-9, 12-1, 16-1, 18-27
 - aliases 16-9 to 16-11
 - remote connections 17-8
 - specifying 17-4, 17-5
 - API calls 13-2
 - batch moves 20-21, 20-22
 - closing database connections 16-7
 - connecting to databases 13-5, 51-5
 - deploying 11-4, 11-11
 - determining table types 20-3
 - direct calls 18-29
 - driver names 17-5
 - help documentation 21-3
 - multi-table queries 21-14
 - one- and two-tiered applications 13-2 to 13-10
 - opening database connections 16-6
 - private directories 16-13
 - remote servers and 17-7, 17-8
 - retrieving data 18-26, 21-3, 21-13, 21-16
 - testing associations 16-8
 - updating data 25-21, 25-23
 - Web applications and 11-7
- boundaries
 - data ranges 20-14
- bounding rectangles 7-10
- .BPL files 9-1, 11-3
- briefcase model 13-17
- brokering connections 14-22
 - See also* Smart Agents
- Brush property 2-22, 7-3, 7-7, 36-3
- BrushCopy method 36-3, 36-7
- brushes 7-7 to 7-9, 40-5
 - bitmap property 7-8
 - changing 40-7
 - colors 7-7
 - styles 7-8
- Business Object Broker 11-5, 14-21
- business rules 14-2
 - data modules 2-34
- ButtonAutoSize property 27-10
- buttons 2-15 to 2-16
 - adding to toolbars 5-29 to 5-30, 5-31
 - assigning glyphs to 5-29
 - disabling on toolbars 5-32
 - grid columns and 26-21
 - navigator 26-29
 - toolbars and 5-28
- ButtonStyle property 26-20, 26-21
- By Reference Only, type library property 50-11
- ByteType 3-27

C

- .CAB files 48-19
- cabinets (defined) 48-19
- CacheBlobs property 18-30
- cached updates 18-29, 25-1
 - applying 25-4
 - Borland Database Engine support 13-9
 - canceling 25-7 to 25-8
 - checking status 25-10 to 25-11
 - client datasets and 25-3
 - enabling/disabling 25-3
 - error handling 25-23 to 25-26
 - caution 25-23
 - fetching records 25-3
 - overview 25-1 to 25-3, 25-22
 - pending 25-3
 - queries and 21-17
 - record type constants 25-9
 - undeleting records 25-8 to 25-10
- CachedUpdates property 18-29, 25-3
- caching resources 36-2

- caching threads 30-14
 - calculated fields 18-8, 18-26, 19-6
 - assigning values 19-9
 - client datasets 24-8 to 24-9
 - defining 19-8
 - lookup fields and 19-10
 - calendar components 2-18
 - calendars 41-1 to 41-12
 - adding dates 41-4 to 41-9
 - defining properties and events 41-2, 41-6, 41-10
 - making read-only 42-2 to 42-4
 - moving through 41-9 to 41-12
 - resizing 41-4
 - selecting current day 41-9
 - callbacks
 - multi-tiered
 - applications 14-17
 - limits 14-10
 - CanBePooled 51-6
 - Cancel method 18-6, 18-7, 18-23, 23-5, 23-28
 - Cancel property 2-15
 - canceling cached updates 25-7 to 25-8
 - CancelRange method 20-15
 - CancelUpdates method 18-29, 25-8
 - CanModify property
 - data grids 26-25
 - data-aware controls 26-4
 - datasets 18-7
 - queries 21-16
 - tables 20-5
 - Canvas property 2-22, 31-7
 - canvases 31-7, 36-1, 36-3
 - adding shapes 7-10 to 7-11, 7-13
 - common properties, methods 7-3
 - default drawing tools 40-5
 - displaying text 7-23
 - drawing lines 7-5, 7-9 to 7-10, 7-26 to 7-27
 - changing pen width 7-5
 - event handlers 7-24
 - drawing vs. painting 7-4
 - overview 7-1 to 7-3
 - palettes 36-4 to 36-5
 - refreshing the screen 7-2
 - Caption property 2-12
 - column headers 26-22
 - decision grids 27-12
 - group boxes and radio groups 2-19
 - invalid entries 5-17
 - labels 2-20
 - cascaded deletes 15-3
 - cascaded updates 15-3
 - case sensitivity
 - comparisons 18-19
 - indexes 13-15, 24-7
 - CaseInsensitive property 13-15
 - case-sensitive sorts 20-10, 24-7
 - cbsAuto constant 26-20
 - CDAudio disks 7-31
 - CellDrawState function 27-12
 - CellRect method 2-21
 - cells (grids) 2-21
 - Cells function 27-12
 - Cells property 2-22
 - CellValueArray function 27-12
 - CGI programs 4-10, 29-4, 29-5
 - creating 29-6
 - debugging 29-28
 - change log 24-4, 24-20, 24-25
 - saving changes 24-5
 - undoing changes 24-5
 - Change method 42-10
 - ChangedTableName property 20-23
 - CHANGEINDEX 24-6
 - changes
 - canceling 18-23
 - Char data type 3-23, 10-2
 - character sets 3-26, 10-2, 10-2 to 10-3
 - 2-byte conversions 10-2
 - ANSI 10-2
 - default 10-2
 - double byte 10-2
 - international sort orders 10-9
 - OEM 10-2
 - character types 3-23, 10-2
 - characters 33-2
 - queries and special 21-6
 - CharCase property 2-12
 - Chart Editing dialog box 27-16
 - Chart FX 11-3
 - check boxes 2-16, 26-2
 - checking 26-15
 - data-aware 26-14
 - CHECK constraint 15-10
 - Checked property 2-16
 - check-list boxes 2-17
 - CheckOpen method 18-28
 - child controls 2-11
 - Chord method 7-4
 - circles
 - drawing 40-9
 - circular references 5-2
 - class factories 44-5, 44-6
 - class fields 40-3
 - declaring 40-5
 - naming 34-2
 - class pointers 32-9
 - classes 31-2, 31-3, 32-1, 33-2
 - abstract 31-3
 - accessing 32-4 to 32-6, 40-6
 - ancestor 32-3
 - creating 32-1
 - default 32-3
 - defining 31-11, 32-2
 - static methods and 32-7
 - virtual methods and 32-8
 - derived 32-8
 - deriving new 32-2, 32-8
 - descendant 32-3, 32-8
 - hierarchy 32-3
 - inheritance 32-7
 - instantiating 32-2
 - passing as parameters 32-9
 - properties as 33-2
 - property editors as 38-7
 - protected part 32-5
 - public part 32-6
 - published part 32-6
 - Clear method 19-17, 21-7
 - string lists 2-31, 2-32
 - ClearSelection method 6-9
 - click events 7-23, 7-24, 34-1, 34-2, 34-7
 - Click method 34-2
 - overriding 34-6, 41-11
 - client applications
 - architecture 14-4
 - as Web server
 - applications 14-28
 - cached updates and 25-1
 - CORBA 28-2, 28-11 to 28-13
 - creating 14-18 to 14-25
 - database 17-1
 - interfaces 30-2
 - multi-tiered 14-1, 14-2, 14-4
 - reconciling updates 24-21 to 24-22
 - updating records 24-20 to 24-22
 - network protocols 17-7
 - refreshing records 24-22 to 24-23
 - retrieving data 21-1, 21-3

- sockets and 30-1
- supplying queries 15-4, 24-17
- thin 14-2, 14-28
- transactions 13-8
- user interfaces 12-9, 14-1
- client connections 30-2, 30-3
 - accepting requests 30-7
 - opening 30-6
 - port numbers 30-5
- client datasets 12-15, 14-3, 24-1 to 24-25
 - adding indexes 24-8
 - aggregating data 24-9 to 24-12
 - cached updates and 25-3
 - calculated fields 24-8 to 24-9
 - constraints 24-3
 - copying data 24-12
 - copying existing tables 13-16
 - creating 13-14, 13-15
 - creating tables 13-15, 24-24
 - deleting indexes 24-7
 - editing 24-4
 - flat-file data 24-24 to 24-25
 - grouping data 24-7
 - indexes 13-17, 24-6 to 24-8
 - adding 24-6
 - limiting records 24-2, 24-16
 - loading data 13-16
 - master/detail
 - relationships 24-3
 - merging changes 24-25
 - navigation 24-2
 - parameters 24-15 to 24-16
 - providers and 24-14 to 24-23
 - reading and writing 24-1
 - read-only 24-4
 - saving changes 24-5
 - saving data 13-16
 - saving edits 13-16
 - sharing data 24-13
 - specifying providers 24-14
 - switching indexes 24-7
 - undoing changes 24-5
- client requests 29-3 to 29-4
- client sockets 30-3, 30-5 to 30-6
 - assigning hosts 30-4
 - connecting to servers 30-8
 - error messages 30-8
 - event handling 30-8
 - identifying servers 30-6
 - properties 30-6
 - requesting services 30-5
 - using threads 30-11
- Windows socket objects 30-6
- client/server applications 4-9
- ClientExecute method
 - TServerClientThread 30-13
- clients *See* client applications
- ClientType property 30-10, 30-11
- Clipboard 6-8, 6-9, 26-10
 - clearing selection 6-9
 - formats
 - adding 38-12, 38-14
 - graphics and 7-20 to 7-22
 - graphics objects 7-3, 26-10
 - testing contents 6-10
 - testing for images 7-21
- Clipbrd unit 6-8
- CloneCursor method 24-13
- cloning cursors 24-13
- Close method
 - database connections 16-7
 - datasets 18-5
 - queries 21-7
 - sessions 16-5
 - tables 20-4
- CloseDatabase method 16-7
- CLSIDs 44-5, 44-6, 44-12
- CM_EXIT message 42-11
- CMExit method 42-11
- CoClasses 44-6
 - adding to type
 - libraries 50-33
 - CLSIDs 44-6
 - instantiating 44-6
 - Type Library editor 50-16 to 50-17, 50-28
- code 35-4
 - optimizing 7-14
- Code editor
 - displaying 38-14
 - event handlers and 2-25
 - opening packages 9-8
- Code Insight
 - templates 4-2
- code pages 10-2
- code signing 48-19
- Code Signing page (Web deployment) 48-23
- ColCount property 26-28
- color depths 11-7
 - programming for 11-9
- color grids 7-5
- Color property 2-11, 2-22
 - brushes 7-7
 - column headers 26-22
 - data grids 26-21
- decision grids 27-12
- pens 7-5
- colors
 - internationalization and 10-8
 - pens 7-5
- Cols property 2-22
- column headers 2-20, 26-18, 26-22
- columns 2-21, 26-16
 - accessing 19-2
 - assigning values 26-18, 26-20
 - changing values 26-25
 - decision grids 27-11
 - default state 26-17, 26-22
 - including in HTML
 - tables 29-23
 - persistent 26-16, 26-18
 - adding buttons 26-21
 - creating 26-19 to 26-22
 - deleting 26-17, 26-19, 26-20
 - inserting 26-19
 - reordering 26-20, 26-26
 - properties 26-17, 26-18, 26-21
 - resetting 26-22
- Columns editor
 - creating persistent
 - columns 26-19
 - defining pick lists 26-20
 - deleting columns 26-20
 - reordering columns 26-20
- Columns property 2-17, 26-19
 - grids 26-16
 - radio groups 2-19
- ColWidths property 2-21, 6-14
- COM 44-1
 - applications 44-16
 - distributed 4-11
 - parts 44-2
 - clients 44-3, 44-8
 - CORBA vs. 28-1
 - extensions 44-2, 44-8 to 44-16
 - comparison of
 - technologies 44-9
 - interfaces 44-3 to 44-4
 - adding to type
 - libraries 50-32
 - automation objects 47-6 to 47-8
 - defined 44-2
 - deriving 44-4
 - dispatch identifiers 47-7
 - dual interfaces 47-6
 - implementing 44-5
 - interface pointer 44-4

- marshaling 44-7
- optimizing 44-14
- reference counting 44-4
- type information 44-12
- overview 44-1
- proxy 44-7
- server types 44-5
- specification 44-1, 44-2
- technologies 44-9
- threads and 8-11
- wizards 44-16, 44-17, 45-1, 45-2
- COM interfaces, raising exceptions 50-10
- COM library 44-2
- COM objects 44-3, 44-5 to 44-8
 - creating 45-2 to 45-6
 - defined 44-3
 - instanting 45-3
 - lifetime management 3-16
 - registering 45-6
 - testing 45-6
 - threading models 45-3 to 45-6
 - type checking 44-12, 44-14
 - type information 50-2
 - updating 44-6
 - wizard 45-2
- COM servers 44-3, 44-5 to 44-8, 44-17
 - combined COM and CORBA servers 28-4
 - in-process 44-6
 - out-of-process 44-6
 - remote 44-6
- combo boxes 2-17, 26-2, 26-11, 26-12
 - data-aware 26-12
 - owner-draw 6-11
 - measure-item events 6-14
 - variant styles 6-12
- COMCTL32.DLL 5-28
- CommandCount property 23-9
- Commands property 23-9
- CommandText property 23-18, 23-19, 23-26, 23-27, 24-17
- CommandTimeout property 23-7, 23-28
- CommandType property 23-19, 23-26, 23-27
- Commit method 13-7
- committing transactions 13-7
- CommitTrans method 23-11
- CommitUpdates method 18-29, 25-6
- common dialog boxes 2-23, 43-1
 - creating 43-2
 - executing 43-4
- Common Object Request Broker Architecture *See* CORBA
- communication between tiers 14-4, 14-8 to 14-10, 14-19
 - CORBA 14-10, 14-22
 - DCOM 14-8, 14-19
 - HTTP 14-9, 14-21
 - OLE 14-21
 - OLEEnterprise 14-10
 - TCP/IP 14-9, 14-20
- communications 30-1
 - protocols 29-1, 30-2
 - connection components 12-10, 14-4
 - networks 17-7
 - UDP vs. TCP 28-3
 - standards 29-1
 - OMG 28-1
- CompareBookmarks method 18-13
- comparison operators 18-17
- compiler directives
 - package-specific 9-11
 - strings 3-31
- compiler options 4-2
- compile-time errors
 - override directive and 32-8
- component editors 38-12 to 38-15
 - default 38-12
 - registering 38-15
- component interfaces
 - creating 43-3
 - properties, declaring 43-3
- Component palette 2-10
 - adding components 9-7, 38-1, 38-3
 - adding database sessions 16-16
 - creating databases 17-2
 - data-aware controls 26-2
 - frames 5-13
 - pages listed 2-10
- Component palette, adding COM servers to 46-1
- component templates 5-11, 5-12, 32-2
 - and frames 5-13, 5-14
- Component wizard 31-8
- component wrappers 31-4, 43-2
 - initializing 43-3
- components 2-6, 32-1, 33-2
- abstract 31-3
- adding to Component Palette 38-1
- adding to existing unit 31-10
- adding to units 31-10
- changing 39-1 to 39-3
- common properties 2-11 to 2-12
- context menus 38-12, 38-12 to 38-13
 - creating 31-2, 31-8
- custom 2-37, 5-11
- customizing 31-3, 33-1, 34-1
- data-aware 42-1
- data-browsing 42-1 to 42-7
- data-editing 42-7 to 42-11
- decision support 27-1
- dependencies 31-5
- derived classes 31-3, 31-11, 40-2
- double-click 38-12, 38-13 to 38-14
- grouping 2-18 to 2-20
- initializing 33-10, 40-6, 42-6
- installing 2-37, 9-6 to 9-7, 38-19
- interfaces 32-4, 32-5, 43-1
 - design-time 32-6
 - runtime 32-6
- memory management 2-9
- MTS 51-2
- nonvisual 2-27, 31-4, 31-11, 43-2
- online help 38-4
- overview 2-10, 31-1
- ownership 2-9
- packages 38-19
- palette bitmaps 38-3
- problems installing 38-19
- registering 31-11
- registration 38-2
- renaming 2-4 to 2-5
- resizing 2-14
- resources, freeing 43-5
- responding to events 34-6, 34-7, 34-8, 42-6
- standard 2-10
- testing 31-12, 43-6
- ComputerName property 14-20, 14-21
- ConfigMode property 16-10
- configuration modes
 - database sessions 16-10
- connected line segments 7-9, 7-10

- Connected property 17-7, 23-3
- connection components 12-10, 14-3, 14-4, 14-19 to 14-25
- Connection property 23-13, 23-18, 23-19, 23-21, 23-22
- ConnectionObject property 23-4
- connections 4-9
 - See also* communication
 - between tiers
 - client 30-3
 - database 17-1, 17-4 to 17-9, 18-28
 - pooling 14-6, 51-12
 - database servers 17-6, 17-8
 - disabling 16-5
 - disconnecting 16-7, 17-8
 - temporary components and 16-6
 - dropping 14-23
 - temporary database 16-8
 - managing 14-23
 - MTS 51-3
 - network protocols 17-7
 - opening 14-19, 14-23, 16-5, 16-6, 30-6
 - parameters 17-6
 - persistent 16-6, 16-7
 - remote applications
 - unauthorized access 17-6
 - remote servers 17-7 to 17-8
 - session 18-28
 - setting default behavior 16-6
 - setting parameters 17-5, 17-6
 - TCP/IP 30-2 to 30-3
 - terminating 30-8
- ConnectionString
 - property 23-3, 23-7, 23-18, 23-19, 23-21, 23-22
- ConnectionTimeout
 - property 23-7
- ConnectOptions property 23-5
- ConnectTo method 46-3
- consistency
 - transactions 51-7
- console applications 4-3
 - CGI 29-5
- CONSTRAINT constraint 15-10
- ConstraintBroker Manager 11-5
- ConstraintErrorMessage
 - property 19-12, 19-22
- constraints 15-10, 19-21 to 19-22, 24-18 to 24-20
 - client datasets 24-3
 - controls 5-3 to 5-4
 - creating 19-22
 - custom 24-19
 - disabling 24-19
 - importing 15-11
- Constraints property 5-4, 15-11
- constructors 2-9, 31-12, 33-10, 35-3, 41-3, 41-4, 42-6
 - multiple 5-8
 - overriding 39-2
 - owned objects and 40-5, 40-6
- Contains list (packages) 9-7, 9-10, 38-19
- Content method
 - page producers 29-19
- content producers 29-8, 29-17
 - event handling 29-19, 29-20, 29-21
- Content property
 - Web response objects 29-16
- ContentFromStream method
 - page producers 29-19
- ContentFromString method
 - page producers 29-19
- ContentStream property
 - Web response objects 29-16, 29-17
- context menus
 - adding items 38-12 to 38-13
 - Menu designer 5-22
 - toolbars 5-34
- context numbers (Help) 2-21
- controlling Unknown 3-20, 3-21, 3-22
- controls 2-6
 - changing 31-3
 - custom 31-4
 - data-browsing 42-1 to 42-7
 - data-editing 42-7 to 42-11
 - display options 2-11
 - generating ActiveX
 - controls 48-3
 - graphical 36-3, 40-1 to 40-9
 - creating 31-4, 40-2
 - drawing 40-2 to 40-9
 - events 36-7
 - grouping 2-18 to 2-20
 - moving through 2-11
 - owner-draw 6-11, 6-13
 - styles 6-12
 - palettes and 36-4 to 36-5
 - position 2-11
 - receiving focus 31-4
 - repainting 40-7, 40-8, 41-4
 - resizing 36-7, 41-4
 - shape 40-7
 - size 2-11
- wINDOWED 31-3
- ControlType property 27-9, 27-15
- conversions 19-17, 19-18
 - PChar 3-30
 - string 3-30
- cool bars 2-16, 5-28
 - adding 5-33
 - configuring 5-33
 - designing 5-28 to 5-35
 - hiding 5-34
- coordinates
 - current drawing
 - position 7-24
- Copy (Object Repository) 2-36
- CopyFile function 3-36
- CopyFrom function 3-40
- copying
 - datasets 20-21
- CopyMode property 36-3
- CopyRect method 7-4, 36-3, 36-7
- CopyToClipboard method 6-9, 26-10
 - graphics 26-10
- CORBA 28-1 to 28-19
 - COM vs. 28-1
 - connecting to application servers 14-22
 - environment variables 28-17
 - initializing 28-11
 - multi-tiered database applications 14-10
 - overview 28-2 to 28-4
 - standards 28-1
 - threads 8-12
- CORBA applications 4-11, 28-1
 - clients 28-11 to 28-13
 - combined COM and CORBA servers 28-4
 - deploying 28-16 to 28-19
 - overview 28-2 to 28-4
 - servers 28-4 to 28-11
- CORBA Data Module
 - wizard 14-15 to 14-16
- CORBA data modules
 - instanting 14-15
 - threading models 14-16
 - wizard 28-4
- CORBA factories 28-7
- CORBA objects
 - defining interfaces 28-5 to 28-8
 - displaying 28-15
 - hiding 28-15

- instanting 28-5
- threading 28-5
- wizard 28-4
- CorbaBind function 28-13
- CorbaInit unit 28-11
- Count property 2-30
 - TSessionList 16-16
- Create DataSet command 13-14
- Create method 2-9
 - databases 17-3
- Create Submenu command (Menu designer) 5-20, 5-23
- CreateDataSet method 13-15
- CreateFile function 3-36
- CreateSuspended parameter 8-10
- CreateTable method 20-17
- CreateTransactionContextEx example 51-19
- creating
 - datasets 13-14, 24-24
 - tables 13-10, 13-14
- critical sections 8-7
 - warning about use 8-7, 8-8
- crostabs 27-2 to 27-3, 27-10
 - defined 27-2
 - multidimensional 27-3
 - one-dimensional 27-2
 - summary values 27-2, 27-3
- cryptographic digest algorithms 48-25
- Ctrl3D property 2-11
- currency
 - internationalizing 10-9
- currency formats 10-9
- Currency property 19-12
- current record 18-9
 - canceling cached updates 25-8
 - setting 20-7
 - synchronizing 20-24
- cursor 18-9
 - moving 18-10, 20-6, 20-7, 26-7
 - to first row 18-10, 18-11
 - to last row 18-10, 18-12
 - with conditions 18-14
 - queries and 21-15
- CurValue property 19-21, 25-26
- custom components 2-37
- custom controls 31-4
 - libraries 31-4
- custom data formats 19-17
- custom datasets 12-15
- Custom property 14-37

- CustomConstraint
 - property 19-12, 19-22
- customizing components 33-1
- CutToClipboard method 6-9, 26-10
 - graphics 26-10

D

- data
 - accessing 42-1
 - analyzing 12-12
 - changing 18-20, 18-23, 26-4
 - transactions and 13-9
 - default values 19-21, 26-11
 - displaying 19-18
 - current values 26-9
 - disabling display 26-4
 - in grids 26-16, 26-17, 26-21, 26-27
 - multiple datasets 26-31
 - refresh intervals 26-4, 26-5
 - display-only 26-8
 - entering 18-21, 18-22
 - formats,
 - internationalizing 10-9
 - graphing 12-12
 - predefined values 26-12
 - printing 12-15
 - reporting 12-15
 - saving 18-23
 - synchronizing 20-24
 - on multiple forms 26-6, 26-7
- data access components 12-1
 - isolating 12-6
 - threads 8-4
- Data Access page (Component palette) 2-10, 12-1
- data brokers 14-1, 24-14
- data compression
 - TSocketConnection 14-20
- data constraints *See* constraints
- Data Controls page (Component palette) 2-10, 12-11, 26-2
- Data Dictionary 12-4 to 12-5, 19-14 to 19-15
 - constraints 15-11
- data filters 18-8, 18-16 to 18-19, 20-11
 - enabling/disabling 18-16
 - queries vs. 18-16, 21-2
 - setting at runtime 18-18
- data formats
 - assigning 19-14
 - customizing 19-17
 - default 19-16
- data grids 12-12, 26-2, 26-27
 - customizing 26-17 to 26-19
 - default state 26-17
 - restoring 26-22
 - displaying ADT fields 26-22
 - displaying array fields 26-22
 - displaying data 26-16, 26-17, 26-21, 26-27
 - drawing 26-26
 - editing 26-25
 - editing data 26-4
 - entering data 26-20
 - event handling 26-27
 - getting values 26-17, 26-18
 - at runtime 26-18
 - inserting columns 26-19
 - overview 26-16
 - properties 26-17, 26-28
 - removing columns 26-17, 26-19, 26-20
 - reordering columns 26-20, 26-26
 - runtime options 26-24
- data integrity 12-5, 15-10
- data links 20-25, 42-4 to 42-6
 - initializing 42-6
- Data Module Designer 2-33 to 2-34
- data modules 2-33 to 2-34
 - accessing from forms 2-34
 - business rules 2-34
 - creating 2-33
 - databases 17-9
 - editing 2-33
 - remote vs. standard 2-33
 - session components and 16-17
 - Web applications and 29-6, 29-7, 29-8
- data packets 14-16, 15-1
 - application-defined information 15-4, 24-12
 - contents 15-1
 - controlling fields 15-2
 - editing 15-5
 - ensuring unique records 15-3
 - fetching 15-5, 24-17
 - including field properties 15-3
 - limiting client edits 15-4
 - read-only 15-3

- refreshing updated records 15-4
- XML 14-28, 14-30, 14-33
 - fetching 14-33
- Data property
 - client datasets 24-12
- data sources 12-11, 26-5 to 26-8
 - adding 26-6
 - associating with
 - datasets 26-6, 26-7
 - binding to queries 21-10
 - defined 26-5
 - naming 26-6
 - remote servers 17-8
 - updating 26-7
- data state constants 18-3
- data store 23-2
- data types 19-7
 - mapping 20-22
- data/time fields 19-15
- data-aware controls 12-11, 19-18, 26-1, 42-1
 - adding 26-1 to 26-2
 - creating 42-1 to 42-12
 - data-browsing 42-1 to 42-7
 - data-editing 42-7 to 42-11
 - destroying 42-6
 - disabling 18-12
 - disabling display 26-4
 - displaying data 21-16, 26-4 to 26-5
 - current values 26-9
 - in grids 26-16, 26-17, 26-21, 26-27
 - multiple datasets 26-31
 - displaying graphics 26-10
 - editing 18-7, 18-20, 26-3
 - entering data 19-15, 26-11 to 26-14, 26-15
 - grids 12-12
 - inserting records 18-21
 - listed 26-2
 - representing fields 12-11, 26-8
 - responding to changes 42-6
- database applications 12-1
 - architecture 12-6, 14-28
 - BDE-based vs. flat-file 13-14
 - deploying 11-4
 - distributed 4-12
 - flat-file 13-13 to 13-17
 - multi-tiered 14-3 to 14-10
 - scaling 12-6, 13-18
- database architecture 12-6
 - database components 17-1 to 17-3
 - iterating 16-12
 - temporary 16-7, 16-8, 17-2
 - database connections 4-9
 - limiting 14-7
 - pooling 14-6, 51-12
 - warning when using MTS 14-6
 - database drivers 12-2
 - database engines
 - third-party 11-5
 - Database Explorer 4-9, 11-6
 - database management systems 14-1
 - database navigator 18-9, 18-10, 26-2, 26-29 to 26-31
 - buttons 26-29
 - deleting data 18-22
 - editing and 18-21
 - enabling/disabling buttons 26-30
 - help hints 26-31
 - Database Properties editor 17-5
 - viewing connection parameters 17-6
 - Database property 18-28
 - database servers 4-9, 17-6, 21-3
 - DatabaseCount property 16-12
 - DatabaseName property 13-3, 17-4, 18-28, 20-2
 - databases 4-9, 12-1 to 12-5, 13-3, 17-1, 17-9, 42-1
 - access properties 42-5 to 42-6
 - accessing 16-1, 16-6, 20-2, 21-4
 - adding data 18-21 to 18-22, 18-24
 - adding tables 20-17
 - aliases and 17-4, 20-2
 - applying cached updates 25-5
 - associating with sessions 16-2, 16-8, 17-4
 - BDE and 13-5
 - changing data 13-9, 18-20, 18-23
 - choosing 12-2
 - closing 16-5
 - connecting 13-5
 - counting 16-12
 - creating 16-2, 17-2
 - at runtime 17-2
 - data sources and 26-5
 - DatabaseName property and 13-3
 - deleting tables 20-16
 - file-based 12-2
 - generating HTML responses 29-21 to 29-24
 - importing data 20-18
 - in multi-threaded applications 13-4
 - limiting data retrieval 20-11
 - local 12-2
 - logging in 12-3
 - logging into 17-6
 - marking records 18-12 to 18-14
 - multi-tiered models 14-2
 - naming 17-4, 20-2
 - relational 12-1
 - remote 12-2
 - renaming tables 20-16
 - resorting fields 20-10
 - retrieving data 18-16, 19-18, 25-1
 - saving data 18-23
 - security 12-3
 - tables 12-13
 - testing associations 16-8
 - testing status 16-4
 - transactions 12-3 to 12-4, 13-6
 - types 12-2
 - unauthorized access 17-6
 - Web applications and 29-21
- Databases property 16-12
- DataChange method 42-10
- data-entry validation 19-17
- DataField property 26-12, 26-14, 42-5
- DataSet component 18-2
- dataset fields 12-13, 19-26 to 19-27
 - client datasets 24-3
- dataset page producers 29-22
 - converting field values 29-22
- DataSet property 26-6
 - data grids 26-17
 - providers 15-1
- DataSetCount property 17-8, 23-8
- DataSetField property
 - client datasets 24-3
- datasets 12-13, 18-1, 18-2
 - accessing 18-26, 26-7
 - adding records 18-7, 18-21 to 18-22, 18-24

- ADO-based 12-14
- applying cached updates 25-5, 25-6
- associating with sessions 16-2, 17-4
- BDE-based 12-13 to 12-14
- BDE-enabled 18-26
- browsing 18-6
- changing data 18-20, 18-23, 26-4
- closing 16-5, 17-8, 18-3, 18-5
 - on remote servers 16-6
 - without disconnecting 17-8
- copying 20-21
- creating 13-14
- current value 25-26
- custom 12-15
- data sources and 26-5
- decision components 27-4 to 27-7
- default state 18-4
- deleting records 18-22
- editing 18-6, 18-20, 26-7
- event handling 18-25
- filtering records 18-8, 18-16 to 18-19
- getting active 17-8
- HTML documents 29-24
- InterBase direct 12-14 to 12-15
- modes 18-3
- moving through 18-9 to 18-12, 18-19, 26-29
- nested 12-13
- opening 18-3
 - on remote servers 16-6
- posting records 18-23
- previous values 25-10, 25-26
- providers and 15-2
- referencing 25-24
- searching 18-8, 18-14 to 18-16
- states 18-3
 - changing 26-7
- testing status 16-4
- updating 25-11, 25-20, 25-21, 25-24
 - updating multiple 25-5
 - viewing multiple 26-31
- DataSets property 17-8, 23-8
- DataSource component
 - adding 26-6
 - events 26-7
 - properties 26-6 to 26-7
- DataSource property 26-2, 26-14, 26-31
 - data grids 26-17
 - data-aware controls 42-5
 - queries 21-10
- date fields 19-15
 - formatting values 19-16
- dates
 - calendar components 2-18
 - entering 2-18
 - internationalizing 10-9
- DateTimePicker component 2-18
- DAX 44-2
- Day property 41-5
- DB/2 driver
 - deploying 11-5
- dBASE 13-11
- dBASE tables 20-2, 20-3
 - accessing data 20-5, 21-4
 - adding records 18-22
 - creating aliases 16-11
 - DatabaseName 13-3
 - indexes 20-9
 - isolation levels 13-8
 - local transactions 13-9
 - memo fields 26-9, 26-10
 - opening connections 16-6
 - queries 21-16
 - searching 20-6, 20-8
- DBChart component 12-12
- DBCheckBox component 26-2, 26-14
- DBComboBox component 26-2, 26-12
- DBCtrlGrid component 26-2, 26-27 to 26-29
 - properties 26-28
- DBEdit component 26-2, 26-9
- DBGrid component 26-2, 26-16
 - events 26-27
 - properties 26-21, 26-24
- DBGridColumn component 26-16
- DBHandle property 18-28
- DBImage component 26-2, 26-10
- DBListBox component 26-2, 26-11
- DBLocale property 18-28
- DBLookupComboBox component 26-2, 26-12 to 26-14
- DBLookupListBox
 - component 26-2, 26-12 to 26-14
- DBMemo component 26-2, 26-9
- DBMS 14-1
- DBNavigator component 26-2, 26-29 to 26-31
- DBRadioGroup component 26-2, 26-15
- DBRichEdit component 26-10
- DBSession property 18-28
- DBText component 26-2, 26-8
- DCOM 44-7
 - connecting to application server 14-19
 - distributing applications 4-11
 - InternetExpress applications 14-32
 - MTS 51-14
 - multi-tiered applications 14-8
- DCOMCnfg.exe 14-32
- .DCP files 9-2, 9-13
- .DCR files 38-3
- .DCU files 9-2, 9-13
- DDL 23-20, 23-26
- debugging
 - ActiveX controls 45-6, 48-17
 - Microsoft IIS server 29-25
 - MTS objects 51-20
 - Netscape servers 29-27
 - Personal Web server 29-27
 - Web server applications 29-25 to 29-29
- Decision Cube editor 27-7
- Decision Cube page (Component palette) 2-10, 12-12
- decision cubes 27-7 to 27-8
 - design options 27-8
 - dimension settings 27-8, 27-19, 27-20
 - dimensions, opening/closing 27-9
 - displaying data 27-11
 - getting values 27-4, 27-5
 - paged dimensions 27-20
- decision datasets 27-5, 27-6
- decision graphs 27-13 to 27-18
 - changing graph type 27-16
 - creating 27-13
 - current pivot states 27-9
 - customizing 27-15 to 27-17
 - display options 27-15

- overview 27-13
- runtime behaviors 27-19
- setting data series 27-17 to 27-18
- templates 27-16
- decision grids 27-10 to 27-13
 - creating 27-10
 - current pivot states 27-9
 - events 27-12
 - overview 27-11
 - properties 27-12
 - reordering columns/rows 27-11
 - runtime behaviors 27-18
 - viewing data 27-11
- decision pivots 27-9
 - properties 27-10
 - runtime behaviors 27-18
- decision queries
 - getting values 27-6
 - properties 27-7
- Decision Query editor 27-6
 - starting 27-6
- decision sources 27-9
 - events 27-9
 - properties 27-9
- decision support 12-12
- decision support components 27-1, 27-18
 - adding 27-3 to 27-4
 - allocating memory 27-19
 - design options 27-8
 - getting values 27-4 to 27-7
 - overview 27-1 to 27-2
- declarations
 - classes 32-9, 40-5
 - protected 32-5
 - public 32-6
 - published 32-6
 - event handlers 34-5, 34-8, 41-11
 - message handlers 37-4, 37-5, 37-7
 - methods 7-14, 35-4
 - dynamic 32-9
 - public 35-3
 - static 32-7
 - virtual 32-8
 - new component types 32-3
 - properties 33-2, 33-3, 33-3 to 33-6, 33-7, 33-10, 34-8, 40-3
 - stored 33-10
 - user-defined types 40-3
 - variables
 - example 2-7
- DECnet protocol (Digital) 30-1
- default
 - ancestor class 32-3
 - data formats 19-16
 - directive 33-9, 39-3
 - handlers
 - events 34-9
 - message 37-3
 - handling, overriding 34-9
 - property values 33-7
 - changing 39-2, 39-3
 - specifying 33-9 to 33-10
 - reserved word 33-7
 - values 19-21, 26-11
- default project options 4-2
- Default property 2-15
 - action items 29-11
- DEFAULT_ORDER 24-6
- DefaultColWidth property 2-21
- DefaultDrawing
 - property 26-26, 26-27
- DefaultExpression
 - property 19-21
- DefaultHandler method 37-3
- DefaultRowHeight
 - property 2-21
- delegation 34-1
- Delete command (Menu designer) 5-23
- Delete method 18-7, 18-22
 - string lists 2-31, 2-32
- DELETE statements 21-12, 21-13, 25-11
- Delete Templates command (Menu designer) 5-23, 5-25
- Delete Templates dialog
 - box 5-25
- DeleteAlias method 16-11
- DeleteFile function 3-33
- DeleteFontResourc
 - function 11-10
- DeleteSQL property 25-11
- DeleteTable method 20-16
- Delphi
 - ActiveX framework (DAX) 44-2
- DELPHI32.DRO 2-35
- delta packets 15-5
 - editing 15-6, 15-6 to 15-7
 - XML 14-33, 14-34 to 14-35
- Delta property 24-20
- \$DENYPACKAGEUNIT
 - compiler directive 9-11
- DEPLOY.TXT 11-4, 11-5, 11-10, 11-11
- deploying
 - ActiveX controls 11-3
 - applications 11-1
 - Borland Database Engine 11-4
 - CORBA 28-16 to 28-19
 - database applications 11-4
 - DLL files 11-3
 - fonts 11-9
 - general applications 11-1
 - MIDAS applications 11-5
 - package files 11-3
 - Web applications 11-7
- dereferencing object
 - pointers 32-9
- deriving classes 32-8
- descendant classes 2-6, 32-3
 - redefining methods 32-8
- DescFields property 13-15
- DESIGNONLY compiler
 - directive 9-11
- design-time interfaces 32-6
- design-time license
 - ActiveForms 48-5, 48-7
 - ActiveX controls 48-5, 48-7
- design-time packages 9-1, 9-5 to 9-7
- destination datasets, defined 20-20
- Destroy method 2-9
- destructors 2-9, 35-3, 42-6
 - owned objects and 40-5, 40-6
- detaching from databases 17-8
- detail datasets 20-24 to 20-26
 - fetch on demand 15-3
- detail forms
 - cached updates and 25-6
- developer support 1-3
- device contexts 7-1, 31-7, 36-1
- device-independent graphics 36-1
- DeviceType property 7-30
- .DFM files 10-9, 33-8
 - generating 10-12
- 2-4
- diacritical marks 10-9
- dialog boxes 43-1 to 43-6
 - common 2-23
 - creating 43-1
 - internationalizing 10-8, 10-9
 - multipage 2-19
 - property editors as 38-9
 - setting initial state 43-1
 - Windows common 43-1
 - creating 43-2

- executing 43-4
 - Dialogs page (Component palette) 2-10
 - .DIB files 26-10
 - digital audio tapes 7-31
 - DII 28-11, 28-12
 - Interface Repository 28-8
 - non-Delphi servers 28-11
 - parameters 28-14
 - DimensionMap property 27-7
 - Dimensions property 27-12
 - directives
 - \$H compiler 3-24, 3-31
 - \$P compiler 3-32
 - \$V compiler 3-32
 - \$X compiler 3-32
 - default 33-9, 39-3
 - dynamic 32-9
 - override 32-8
 - protected 34-5
 - public 34-5
 - published 33-2, 34-5, 43-3
 - stored 33-10
 - virtual 32-8
 - directories
 - temporary files 16-13
 - directory locations
 - ActiveX deployment 48-20
 - directory service 28-3
 - dirty reads 13-7
 - DisableCommit 51-9
 - DisableConstraints
 - method 24-19
 - DisableControls method 26-4
 - DisabledImages property 5-32
 - disabling cached updates 25-3
 - Disconnect method 46-3
 - disconnected model 13-17
 - dispatch interfaces 47-7 to 47-8
 - attributes (Type Library editor) 50-14 to 50-15
 - identifiers 47-7
 - type compatibility 47-9
 - Type Library editor 50-28
 - Dispatch method 37-3, 37-4
 - dispIDs 44-12, 47-7
 - binding to 47-8
 - dispinterfaces 14-24, 47-6
 - Dispinterfaces, in type libraries 50-14
 - DisplayFormat property 19-2, 19-12, 19-16, 26-26
 - DisplayLabel property 19-12, 26-18
 - display-only data 26-8
 - DisplayWidth property 19-2, 19-12, 26-17
 - distributed applications 4-9, 4-9 to 4-12
 - COM 4-11
 - CORBA 4-11, 28-1
 - database 4-12
 - MTS 4-11
 - multi-threaded 8-11
 - distributed data processing 14-2
 - distributed objects
 - COM 4-11
 - CORBA 4-11, 28-1
 - threads 8-11
 - DLLs
 - COM servers 44-6
 - creating 4-8
 - embedding in HTML 29-18
 - HTTP servers 29-4
 - installing 11-3
 - internationalizing 10-11, 10-12
 - packages 9-1, 9-2
 - DML 23-20, 23-26
 - .DMT files 5-24, 5-25
 - docking 6-4
 - documentation
 - ordering 1-3
 - double byte character set 10-2
 - double-clicks
 - components 38-12
 - responding to 38-13 to 38-14
 - Down property 2-16
 - speed buttons 5-30
 - .DPK files 9-2, 9-7
 - .DPL files 9-2, 9-13
 - drag cursors 6-2
 - drag object 6-3
 - drag-and-dock 2-12, 6-4 to 6-6
 - drag-and-drop 2-12, 6-1 to 6-4
 - customizing 6-3
 - DLLs 6-4
 - events 40-2
 - getting state information 6-3
 - mouse pointer 6-4
 - DragCursor property 2-12
 - DragMode property 2-12, 6-1
 - grids 26-26
 - draw grids 2-21
 - Draw method 7-4, 36-3, 36-7
 - drawing modes 7-27
 - drawing tools 36-1, 36-7, 36-8, 40-5
 - assigning as default 5-30
 - changing 7-12, 40-7
 - handling multiple in an application 7-11
 - testing for 7-11, 7-12
 - DrawShape 7-15
 - drill-down forms 12-12
 - dprintf unit 12-4
 - driver names 17-5
 - DriverName property 17-4
 - DropConnections method 16-8
 - drop-down lists 26-12, 26-18
 - assigning values 26-20
 - drop-down menus 5-19 to 5-20
 - DropDownCount property 2-17
 - DropDownMenu property 5-34
 - DropDownRows
 - property 26-14, 26-21
 - dropped connections
 - temporary databases 16-8
 - DsgnIntf unit 38-7
 - dsSetKey constant 18-8
 - dual interfaces
 - auto-marshaling 44-16
 - MTS 51-16
 - optimizing 44-14
 - parameters 47-9
 - type compatibility 47-9
 - durability
 - resource dispensers 51-11
 - transactions 51-7
 - dynamic binding 14-24, 28-11
 - CORBA 28-4, 28-12
 - DII 28-4
 - dynamic directives 32-9
 - dynamic invocation interface
 - See DII
 - dynamic memory 2-33
 - dynamic methods 32-8
- ## E
-
- EAbort 3-12
 - early binding 14-24, 28-11
 - Automation 44-14
 - EDBEngineError type 25-25
 - edit control 2-12
 - edit controls 2-12 to 2-13, 6-6, 26-2, 26-9
 - multi-line 26-9
 - rich edit formats 26-10
 - selecting text 6-8
 - Edit method 18-6, 18-20, 38-9, 38-10
 - edit mode 18-20
 - canceling 18-21
 - EditFormat property 19-2, 19-12, 19-16, 26-26

- editing data 18-6
- editing properties
 - array 33-2
 - field objects 19-12
- EditKey method 20-6, 20-8
- EditMask property 19-12, 19-15
- EditRangeEnd method 20-15
- EditRangeStart method 20-15
- Ellipse method 7-4, 7-10, 36-3
- ellipses
 - drawing 7-10, 40-9
- ellipsis (...)
 - buttons in grids 26-21
- Embed HTML tag (<EMBED>) 29-18
- EmptyStr variable 3-29
- EmptyTable method 20-16
- EnableCommit 51-9
- EnableConstraints
 - method 24-19
- EnableControls method 26-4
- Enabled property
 - action items 29-11
 - data sources 26-7
 - data-aware controls 26-3, 26-5
 - menus 5-26, 6-10
 - speed buttons 5-30
- enabling cached updates 25-3
- encapsulation 2-2
- encryption
 - TSocketConnection 14-20
- endpoints 30-5
- EndRead method 8-7
- EndWrite method 8-7
- enumerated types 33-2, 40-3
 - adding to type
 - libraries 48-11, 48-12, 50-33
 - constants vs. 7-12
 - declaring 7-11
 - Type Library editor 50-18, 50-29
- environment variables
 - CORBA 28-17
- EOF marker 3-40
- Eof property 18-10, 18-11
- EPasswordInvalid 3-13
- EReadError 3-38
- error messages 25-25
 - internationalizing 10-9
- ErrorAddr variable 3-13
- errors
 - batch moves 20-23
 - cached updates 25-23 to 25-26
 - caution 25-23
 - data providers 15-9
 - override directive and 32-8
 - sockets 30-8
- event handlers 2-4, 2-24 to 2-27, 19-17, 31-6, 34-2, 42-6
 - associating with events 2-25
 - declarations 34-5, 34-8, 41-11
 - default, overriding 34-9
 - defined 2-24
 - deleting 2-27
 - displaying the Code editor 38-14
 - drawing lines 7-24
 - empty 34-8
 - locating 2-25
 - menus 2-26 to 2-27, 6-11
 - as templates 5-26
 - methods 34-3, 34-5
 - overriding 34-6
 - parameters 34-3, 34-7, 34-8, 34-9
 - notification events 34-7
 - passing parameters by reference 34-9
 - pointers 34-2, 34-3, 34-8
 - responding to button clicks 7-12
 - Sender parameter 2-26
 - shared 2-25 to 2-27
 - sharing 7-14
 - sharing code among 7-14
 - types 34-3, 34-7 to 34-8
 - writing 2-6, 2-24, 2-25
- event objects 8-9
- events 2-24 to 2-27, 26-5, 31-6, 34-1 to 34-9
 - accessing 34-5
 - ActiveX controls 48-9, 48-10
 - application-level 5-3
 - associating with handlers 2-25
 - data grids 26-27
 - data sources 26-7
 - datasets 18-25
 - decision cubes 27-7
 - decision grids 27-12
 - decision sources 27-9
 - default 2-25
 - defining new 34-6 to 34-9
 - field objects 19-16
 - graphical controls 36-7
 - help with 38-4
 - implementing 34-2, 34-4
 - inherited 34-5
 - message handling and 37-3, 37-5
 - mouse 7-22 to 7-25
 - testing for 7-25
 - MTS objects 51-15
 - naming 34-8
 - objects and 2-8
 - responding to 34-6, 34-7, 34-8, 42-6
 - retrieving 34-3
 - shared 2-26
 - signalling 8-9
 - standard 34-4, 34-4 to 34-6
 - timeout 8-10
 - update objects 25-22 to 25-23
 - waiting for 8-9
 - XML brokers 14-34
- Events, ActiveX controls 46-2, 46-5
- Events, in Active Server Objects 49-3
- events, renaming by mistake 50-32, 50-34
- EWriteError 3-38
- examples
 - service applications 4-6
- Exception 3-13
- exception handling 3-1 to 3-13
 - creating a handler 3-7
 - declaring the object 3-13
 - default handlers 3-9
 - executing cleanup code 3-2
 - flow of control 3-2
 - overview 3-1 to 3-13
 - protecting blocks of code 3-1
 - protecting resource allocations 3-4
 - resource protection
 - blocks 3-5
 - scope 3-8
 - statements 3-7
 - TApplication 3-11
- exceptions 3-1 to 3-13, 35-2, 37-3, 43-5
 - classes 3-9
 - component 3-11
 - datasets 25-25
 - handling 3-2
 - instances 3-8
 - nested 3-3
 - raising 3-13
 - re-raising 3-10
 - responding to 3-2
 - RTL 3-5, 3-6
 - silent 3-12

- user-defined 3-12
- exceptions, raising in COM
 - interfaces 50-10
- exclusive locks 20-5
- Exclusive property 20-5
- ExecProc method 22-5, 23-23
- ExecProc property 23-23
- ExecSQL method 21-12, 21-13
 - update objects 25-19
- executable files
 - COM servers 44-6
 - internationalizing 10-11, 10-12
- Execute method 2-23, 8-3, 23-27, 43-4
 - TBatchMove 20-23
 - threads 30-12
- ExecuteTarget method 5-39
- executing queries 21-12 to 21-13
 - from text files 21-8
 - update objects 25-19
- executing stored
 - procedures 22-5
- Expanded property 26-22
 - TColumn 26-22
- Expose As CORBA Object
 - command 28-4
- expressions 19-21

F

- Fetch Params command 24-15
- FetchAll method 18-29, 25-3
- fetching records 25-3
- fetch-on-demand 24-18
- FetchParams method 24-15
- field attributes 12-4 to 12-5, 19-14 to 19-15
 - removing 19-15
- field definitions 20-17
- field lists 19-4, 20-10
 - viewing 19-5, 19-6
- field names 19-7
- field objects 19-1
 - adding 19-1 to 19-5, 19-6
 - assigning values 19-20
 - deleting 19-11
 - display-only 19-7
 - dynamic vs. persistent 19-3, 19-4
 - events 19-16
 - properties 19-3, 19-12 to 19-15
 - runtime 19-13
- field types 19-1, 19-2
 - converting 19-17, 19-18

- overriding 19-16
- specifying 19-7
- FieldByName method 19-20, 20-13
- FieldCount property
 - persistent fields 26-18
- FieldKind property 19-12
- FieldName property 14-36, 19-7, 19-12
 - data grids 26-20, 26-22
 - decision grids 27-12
 - persistent fields 26-18
- fields 19-1
 - abstract data types 19-23 to 19-28
 - activating 19-17
 - adding to forms 7-25 to 7-26
 - assigning values 18-24
 - changing values 26-4
 - checking current
 - values 19-21
 - current value 25-26
 - databases 42-5, 42-6
 - default values 19-21
 - defining 19-7, 19-8, 19-9
 - displaying values 19-18, 26-12
 - entering data 18-21, 18-22, 19-15, 26-11 to 26-14, 26-15
 - getting 19-4, 19-5
 - message records 37-2, 37-4, 37-6
 - mutually-exclusive
 - options 26-2
 - persistent columns
 - and 26-18
 - previous values 25-10, 25-26
 - read-only 26-3
 - representing 26-8
 - resorting 20-10
 - searching specific 20-8
 - setting data limits 19-21 to 19-22
 - sharing properties 19-14
 - updating values 26-3
- Fields editor 2-34, 19-4
 - applying field
 - attributes 19-14
 - creating persistent
 - fields 19-5
 - defining attribute sets 19-14
 - deleting field objects 19-11
 - removing attribute sets 19-15
 - reordering columns 26-26
- Fields property 19-20

- file I/O
 - types 3-36
- file lists
 - dragging items 6-2, 6-3
 - dropping items 6-3
- file streams
 - changing the size of 3-40
 - creating 3-37
 - end of marker 3-40
 - exceptions 3-38
 - file I/O 3-37 to 3-40
 - getting a handle 3-36
 - opening 3-37
 - portable 3-36
 - TMemoryStream 3-37
 - VCL streaming 3-37
- FileAge function 3-35
- FileExists function 3-33
- FileGetDate function 3-35
- FileName property
 - client datasets 13-17
- files 3-32 to 3-40
 - attributes 3-35
 - copying 3-36
 - copying bytes from 3-40
 - date-time routines 3-35
 - deleting 3-33
 - finding 3-33
 - graphics 7-18 to 7-20, 36-4
 - handles 3-36, 3-38
 - incompatible types 3-36
 - manipulating 3-33 to 3-36
 - modes 3-37
 - position 3-39
 - reading from 3-38
 - renaming 3-35
 - resource 5-27 to 5-28
 - routines
 - date-time routines 3-35
 - runtime library 3-33
 - Windows API 3-36
 - seeking 3-39
 - sending over the Web 29-17
 - size 3-39
 - strings 3-39
 - temporary 16-13
 - types
 - I/O 3-36
 - text 3-36
 - typed 3-36
 - untyped 3-36
 - working with 3-32 to 3-40
 - writing to 3-38
- file streams 3-37 to 3-40
- FileSetDate function 3-35

- fill patterns 7-7, 7-8
 - FillRect method 7-4, 36-3
 - Filter property 18-16, 18-17
 - Filtered property 18-16
 - FilterOptions property 18-19
 - filters 18-8, 18-16 to 18-19, 20-11
 - client datasets 24-2
 - enabling/disabling 18-16
 - queries vs. 18-16, 21-2
 - setting at runtime 18-18
 - finally reserved word 36-6, 43-5
 - FindClose procedure 3-33
 - FindDatabase method 16-8
 - FindFirst function 3-33
 - FindFirst method 18-19
 - FindKey method 18-8, 20-6, 20-7
 - caution for using 20-5
 - EditKey vs. 20-8
 - FindLast method 18-19
 - FindNearest method 18-8, 20-6, 20-7
 - caution for using 20-5
 - FindNext function 3-33
 - FindNext method 18-19
 - FindPrior method 18-19
 - FindResourceHInstance function 10-11
 - FindSession method 16-16
 - First Impression 11-3
 - First method 18-10
 - FixedColor property 2-21
 - FixedCols property 2-21
 - FixedOrder property 2-16, 5-33
 - FixedRows property 2-21
 - FixedSize property 2-16
 - flags 42-3
 - flat files 24-24 to 24-25
 - loading 13-16, 24-24
 - nested tables and 24-3
 - saving 13-16, 24-25
 - flat-file applications 13-13 to 13-17
 - memory 13-13
 - FlipChildren method 10-6
 - FloodFill method 7-4, 36-3
 - fly-by help 2-21
 - fly-over help 26-31
 - focus 19-17, 31-4
 - moving 2-14
 - FocusControl method 19-17
 - FocusControl property 2-20
 - Font property 2-11, 7-3, 36-3
 - column headers 26-22
 - data grids 26-22
 - memo fields 26-10
 - fonts 11-9
 - height of 7-4
 - Footer property 29-24
 - FOREIGN KEY constraint 15-11
 - foreign translations 10-1
 - form linking 5-2
 - Format property 27-12
 - FormatCurr function 19-16
 - FormatDateTime function 19-16
 - FormatFloat function 19-16
 - formatting data 19-14, 19-16
 - custom formats 19-17
 - international applications 10-9
 - forms
 - accessing from other forms 2-7
 - adding fields to 7-25 to 7-26
 - adding to projects 2-5, 5-1 to 5-2
 - adding unit references 5-2
 - as components 43-1
 - as new object types 2-2 to 2-4
 - creating at runtime 5-6
 - displaying 5-5
 - display-only data 26-8
 - drill down 12-12
 - global variable for 5-5
 - instantiating 2-3
 - linking 5-2
 - main 5-1
 - master/detail tables 12-12, 20-24 to 20-26
 - memory management 5-5
 - modal 5-5
 - modeless 5-5, 5-6
 - navigating among controls 2-11
 - passing arguments to 5-7 to 5-8
 - querying properties
 - example 5-8
 - referencing 5-2
 - retrieving data from 5-8 to 5-11
 - scrolling regions 2-19
 - sharing event handlers 7-14
 - synchronizing data 20-24
 - on multiple 26-6, 26-7
 - using local variables to create 5-7
 - Formula One 11-3
 - FoxPro 13-11
 - FoxPro tables 13-8, 20-5
 - isolation levels 13-8
 - local transactions 13-9
 - FrameRect method 7-4
 - frames 5-11, 5-12 to 5-14
 - and component templates 5-13, 5-14
 - graphics 5-14
 - resources 5-14
 - sharing and distributing 5-14
 - Free method 2-9
 - free threading 45-5
 - FreeBookmark method 18-13
 - freeing resources 43-5
 - functions 31-6
 - events and 34-3
 - graphics 36-1
 - naming 35-2
 - reading properties 33-6, 38-8, 38-10
 - Windows API 31-3, 36-1
- ## G
-
- \$G compiler directive 9-11, 9-12
 - GDI applications 31-7, 36-1
 - Generate Event Support Code 47-3
 - geometric shapes
 - drawing 40-9
 - GetAliasDriverName method 16-9
 - GetAliasNames method 16-9
 - GetAliasParams method 16-9
 - GetAttributes method 38-10
 - GetBookmark method 18-13
 - GetConfigParams method 16-9
 - GetData method 19-17
 - GetDatabaseNames method 16-9
 - GetDriverNames method 16-9
 - GetDriverParams method 16-9
 - GetFieldByName method 29-13
 - GetFloatValue method 38-8
 - GetIDsOfNames method 47-8
 - GetIndexNames method 20-9
 - GetMethodValue method 38-8
 - GetOptionalParam method 15-4
 - GetOrdValue method 38-8
 - GetPalette method 36-5
 - GetPassword method 16-15
 - GetProcedureNames method 23-10, 23-23
 - GetProperties method 38-10
 - GetSessionNames method 16-16

- GetStoredProcNames
 - method 16-9
 - GetStrValue method 38-8
 - GetTableNames method 16-9, 23-9, 23-20
 - GetValue method 38-8
 - GetVersionEx function 11-10
 - Glyph property 2-15, 5-30
 - GotoBookmark method 18-13
 - GotoCurrent method 20-24
 - GotoKey method 18-8, 20-6
 - caution for using 20-5
 - GotoNearest method 18-8, 20-6, 20-7
 - caution for using 20-5
 - Graph Custom Control 11-3
 - Graphic property 7-17, 7-20, 36-4
 - graphical controls 31-4, 36-3, 40-1 to 40-9
 - bitmaps vs. 40-3
 - creating 31-4, 40-2
 - drawing 40-2 to 40-9
 - events 36-7
 - saving system resources 31-4
 - graphics 26-10, 36-1 to 36-8
 - adding controls 7-16
 - adding to HTML 29-18
 - associating with strings 2-32
 - changing images 7-19
 - complex 36-5
 - containers 36-4
 - copying 7-21
 - deleting 7-21
 - displaying 2-22 to 2-23
 - drawing lines 7-5, 7-9 to 7-10, 7-26 to 7-27
 - changing pen width 7-5
 - event handlers 7-24
 - drawing tools 36-1, 36-7, 36-8, 40-5
 - changing 40-7
 - drawing vs. painting 7-4
 - file formats 7-3
 - files 7-18 to 7-20
 - functions, calling 36-1
 - in frames 5-14
 - internationalizing 10-8
 - loading 7-18, 36-4
 - methods 36-3, 36-4, 36-6
 - copying images 36-7
 - palettes 36-5
 - overview 36-1 to 36-2
 - owner-draw controls 6-11
 - pasting 7-21
 - programming overview 7-1 to 7-3
 - redrawing images 36-7
 - replacing 7-19
 - resizing 7-19, 26-11, 36-7
 - rubber banding
 - example 7-22 to 7-27
 - saving 7-19
 - standalone 36-3
 - storing 36-4
 - string lists 6-12 to 6-13
 - types of objects 7-2 to 7-3
 - graphics boxes 26-2
 - graphics methods
 - palettes 36-5
 - graphics objects
 - threads 8-5
 - GridLineWidth property 2-21
 - grids 2-21 to 2-22, 26-2, 41-1, 41-2, 41-4, 41-10
 - adding rows 18-21
 - color 7-5
 - customizing 26-17 to 26-19
 - data-aware 12-12, 26-27
 - default state 26-17
 - restoring 26-22
 - displaying data 26-16, 26-17, 26-21, 26-27
 - drawing 26-26
 - editing 26-25
 - editing data 26-4
 - entering data 26-20
 - event handling 26-27
 - getting values 26-17, 26-18
 - at runtime 26-18
 - inserting columns 26-19
 - properties 26-17, 26-28
 - removing columns 26-17, 26-19, 26-20
 - reordering columns 26-20, 26-26
 - runtime options 26-24
 - group boxes 2-19
 - Grouped property
 - tool buttons 5-32
 - GroupIndex property 2-16
 - menus 5-27
 - speed buttons 5-30
 - grouping components 2-18 to 2-20
 - grouping speed buttons 5-30
 - GroupLayout property 27-10
 - Groups property 27-10
 - GUIDs 3-18, 44-3
 - generating 3-18
- ## H
- \$H compiler directive 3-24, 3-31
 - Handle property 3-38, 31-3, 31-5, 36-3
 - device context 7-1
 - sockets 30-6, 30-7
 - HandleException 3-11
 - HandleException method 37-3
 - handles
 - resource modules 10-11
 - socket connections 30-6, 30-7
 - HandlesTarget method 5-39
 - HasConstraints property 19-12
 - HasFormat method 6-10, 7-21
 - header controls 2-20
 - Header property 29-24
 - headers
 - HTTP requests 29-2
 - Height property 2-11, 5-3, 26-11
 - TScreen 11-8
 - Help
 - context sensitive 2-21
 - hints 2-21
 - tool-tip 2-21
 - Help Hints 26-31
 - Help systems 38-4
 - database engine 21-3
 - files 38-4
 - keywords 38-5
 - tool buttons 5-34
 - HelpContext property 2-21
 - HelpFile property 2-21
 - heterogenous joins 21-14
 - heterogenous queries 21-14
 - hidden fields 15-3
 - HideSelection property 2-13
 - hierarchy (classes) 32-3
 - Hint property 2-21
 - hints 2-21
 - Hints property 26-31
 - horizontal track bars 2-14
 - HorzScrollBar 2-14
 - host names 30-4
 - IP addresses vs. 30-4
 - Host property
 - client sockets 30-6
 - TSocketConnection 14-20
 - HostName property
 - TCorbaConnection 14-22
 - hosts 14-20, 30-4
 - addresses 30-4
 - URLs 29-2
 - hot keys 2-14
 - HotImages property 5-32

- HotKey property 2-14
- HRESULT 47-9
- HTML commands 29-18
 - database information 29-22
 - generating 29-19
- HTML documents 29-3
 - databases and 29-21
 - dataset page
 - producers 29-22
 - datasets 29-24
 - embedding tables 29-24
- HTTP response messages 29-4
- InternetExpress
 - applications 14-30
 - page producers 29-17 to 29-21
 - stylesheets 14-36
 - table producers 29-23 to 29-24
 - templates 14-35, 14-37 to 14-38, 29-18 to 29-19
- HTML forms 14-36
- HTML tables 29-18, 29-24
 - captions 29-24
 - creating 29-23 to 29-24
 - setting properties 29-23
- HTML templates 14-37 to 14-38, 29-17
 - default 14-35, 14-37
- HTMLDoc property 14-35, 29-19
- HTMLFile property 29-19
- HTML-transparent tags
 - converting 29-17, 29-19
 - parameters 29-18
 - predefined 14-38, 29-18
 - syntax 29-18
- HTTP 29-2
 - application servers and 14-12
 - connecting to application server 14-21
 - message headers 29-1
 - multi-tiered
 - applications 14-9
 - overview 29-3 to 29-4
 - request headers 29-2, 29-13
 - response headers 29-16
 - status codes 29-15
- HTTP request messages *See* request messages
- HTTP response messages *See* response messages
- httsrvr.dll 14-9, 14-21

- hypertext links
 - adding to HTML 29-18

I

- IAppServer interface 14-4, 14-7 to 14-8, 14-16 to 14-18, 14-27
- IB datasets 12-14
- IB queries 12-14
- IB stored procedures 12-14
- IB tables 12-14
- IClassFactory interface 44-5
- IConnectPoint 47-3
- IConnectPointContainer 47-3
- icons 2-22, 36-4
 - graphics object 7-3
 - toolbars 5-31
 - tree views 2-18
- IDataIntercept interface 14-20
- identifiers
 - class fields 34-2
 - dispatch interfaces 47-7
 - binding to 47-8
 - events 34-8
 - invalid 5-17
 - message-record types 37-6
 - methods 35-2
 - property settings 33-6
- ideographic characters 10-2
 - abbreviations and 10-8
 - wide characters and 10-3
- IDispatch interface 44-8, 44-16, 44-18, 47-7 to 47-8
 - automation 44-11
 - identifiers 47-7
 - binding to 47-8
 - marshaling 44-16
 - tracking members 47-7
- IDL (Interface Definition Language) 28-5, 44-12, 44-14, 50-1
- IDL compiler 44-14
- IDL file 28-5
 - exporting from type library 28-7
 - registering 28-8
- IDL files
 - exporting from type library 50-35
- idl2ir 28-9
- IETF protocols and standards 29-1
- IIDs 44-3
- Image HTML tag () 29-18

- ImageIndex property 5-31, 5-33, 5-37
- ImageMap HTML tag (<MAP>) 29-18
- images 2-22, 26-2, 36-3
 - adding 7-16
 - adding control for 6-12
 - adding to menus 5-21
 - brushes 7-8
 - changing 7-19
 - controls for 7-1, 7-16
 - displaying 2-22 to 2-23
 - drawing 40-8
 - erasing 7-21
 - in frames 5-14
 - internationalizing 10-8
 - redrawing 36-7
 - reducing flicker 36-5
 - regenerating 7-2
 - saving 7-19
 - scrolling 7-16
 - tool buttons 5-31
- Images property
 - tool buttons 5-31
- IMalloc interface 3-14
- IMarshal interface 47-8
- IME 10-7
- ImeMode property 10-7
- ImeName property 10-7
- Implementation Repository 28-4
- implements keyword 3-18, 3-19
- implicit transactions 13-5
- \$SIMPLICITBUILD compiler directive 9-11
- Import Type Library command 46-2
- ImportedConstraint property 19-12, 19-22
- \$IMPORTEDDATA compiler directive 9-11
- importing data 20-18
- incremental fetching 14-26, 24-18
- incremental searches 26-14
- Indent property 2-18, 5-30, 5-32, 5-33
- index definitions 20-17
 - client datasets 13-15
- index files 20-9
- Index Files editor 20-9
- Index property 19-13
- index reserved word 41-6
- index-based searches 18-8, 18-15, 18-16, 20-6

- indexes 20-9 to 20-11, 33-8
 - alternative 20-8, 20-9
 - batch moves and 20-21, 20-22
 - client datasets 13-15, 24-6 to 24-8
 - getting 20-9, 20-10
 - grouping data 24-7
 - searching on partial keys 20-8
 - sorting on ranges 20-12, 20-13, 20-14
- IndexFieldCount
 - property 20-10
- IndexFieldNames
 - property 20-8, 20-10
 - IndexName vs. 20-10
- IndexFields property 20-10
- IndexFiles property 20-9
- IndexName property 20-8, 20-9, 20-10
 - IndexFieldNames vs. 20-10
- IndexOf method 2-31
- .INF files 48-19
- INFINITE constant 8-10
- Informix drivers
 - deploying 11-5
- Informix servers 21-4
- Inherit (Object Repository) 2-36
- inheritance 2-2, 2-5
- inherited
 - events 34-5
 - methods 34-6
 - properties 40-2, 41-2
 - publishing 33-2
- inheriting from classes 2-5 to 2-6, 32-7
- INI files 2-32
 - Win-CGI programs 29-5
- initializing
 - components 33-10, 40-6
 - methods 33-10
 - threads 8-2
- in-process servers 44-6
 - ActiveX 44-11
 - for Active Server Pages 49-3
- input controls 2-13
- input focus 19-17, 31-4
- Input Mask editor 19-15
- input method editor 10-7
- input parameters 22-10
- Insert command (Menu designer) 5-23
- Insert From Resource command (Menu designer) 5-23, 5-28
- Insert from Resource dialog box 5-28
- Insert From Template command (Menu designer) 5-23, 5-24
- Insert method 18-7, 18-22
 - Append vs. 18-21
 - menus 5-26
 - strings 2-31
- INSERT statements 21-12, 21-13, 25-11
- Insert Template dialog box 5-24
- inserting records 18-22, 20-21
- InsertObject method 2-32
- InsertRecord method 18-24
- InsertSQL property 25-11
- installation programs 11-2
- InstallShield Express 11-1
 - deploying applications 11-2
 - deploying BDE 11-4
 - deploying MIDAS 11-6
 - deploying packages 11-3
 - deploying SQL Links 11-5
- instances 34-2
- instancing
 - COM objects 45-2, 45-3
 - CORBA data modules 14-15
 - CORBA objects 28-5
 - remote data modules 14-14
- IntegralHeight property 2-17, 26-11
- Integrated Translation Environment 10-12
- integrity
 - violations 20-23
- InterBase driver
 - deploying 11-5
- InterBase page (Component palette) 12-1
- InterBase tables 21-4
- Interface Definition Language (IDL) 28-5, 50-1
- interface keyword 3-14
- Interface Repository 28-4
 - adding interfaces 28-8
 - registering CORBA interfaces 28-8, 28-8 to 28-9
 - removing entries 28-9
 - running 28-8
- interfaces 3-14 to 3-22, 32-4, 32-5, 43-1, 43-3
 - aggregation 3-18, 3-19
 - as operator 3-17
 - Automation 47-6 to 47-8
 - CLSIDs 3-22
- COM 3-22, 44-1, 44-3 to 44-4, 45-2
 - components 3-21
 - controlling Unknown 3-20, 3-21, 3-22
- CORBA 3-22, 28-3, 28-5 to 28-8
 - allowed types 28-6
- Ctrl+Shift+G 3-18
- delegation 3-18
- deriving 3-16
- design-time 32-6
- DII 28-13
- dispatch 47-7
- distributed applications 3-22
- dynamic binding 3-17, 28-4
- Dynamic Invocation Interface 3-22
- dynamic querying 3-16
- early binding 14-24
- example code 3-15, 3-18, 3-20
- extending single inheritance 3-14, 3-15
- IIDs 3-18, 3-21
- implementing 28-7, 44-5
- inner objects 3-19
- internationalizing 10-8, 10-9, 10-12
- IUnknown, implementing 3-16
- language feature 3-14
- late binding 14-24
- lifetime management 3-16, 3-20
- marshaling 3-22
- memory management 3-17, 3-20
- multi-tiered
 - applications 14-7 to 14-8
 - calling 14-24
- naming 28-5
- nonvisual program elements 31-4
- object destruction 3-20
- optimizing code 3-21
- outer objects 3-19
- overview 3-14 to 3-22
- polymorphism 3-15
- procedures 3-16
- properties, declaring 43-3
- reference counting 3-16, 3-17, 3-20 to 3-22
- registering 28-8 to 28-11
- remote data modules 14-16 to 14-18

- reusing code 3-18
- runtime 32-6
- sharing between classes 3-15
- skeletons and 28-2
- stubs and 28-2
- TComponent 3-21
- type libraries and 44-11
- Type Library editor 50-9 to 50-14, 50-27
 - using 3-14 to 3-22
- internal caches 25-1
- InternalCalc fields 19-6, 24-8 to 24-9
- international applications 10-1
 - abbreviations and 10-8
 - converting keyboard input 10-7
 - localizing 10-11
- internationalization 10-1
- Internet Engineering Task Force 29-1
- Internet Information Server (IIS) 49-1
- Internet page (Component palette) 2-10
- Internet standards and protocols 29-1
- InternetExpress 14-28, 14-30 to 14-38
- InternetExpress page (component palette) 14-30
- intranets
 - host names 30-4
 - See also* local networks
- InTransaction property 13-6, 23-11
- Invalidate method 40-8
- Invoke method 47-8
- IObjectContext 51-6
- IObjectControl 51-2
- IP addresses 30-4, 30-6
 - host names 30-4
 - host names vs. 30-4
 - hosts 30-4
 - multi-homed hosts 28-19
 - Smart Agents 28-19
- IPaint interface 3-15
- IPersist interface 3-14
- IProvideClassInfo interface
 - type libraries 44-13
- IProviderSupport interface 15-1
- IPX/SPX protocols 30-1
- irep 28-8
- IRotate interface 3-15
- is reserved word 2-8

- ISAPI applications 4-10, 11-7, 29-5
 - creating 29-6
 - debugging 29-25
 - request messages 29-7
- IsCallerInRole method 14-5
- isolation
 - transactions 51-7
- isolation levels 13-7 to 13-8
 - ODBC drivers 13-8
- IsValidChar method 19-17
- ITE 10-12
- ItemHeight property 2-17
 - combo boxes 26-12
 - list boxes 26-11
- ItemIndex property 2-17
 - radio groups 2-19
- Items property 2-17
 - combo boxes 26-12
 - list boxes 26-11
 - radio controls 26-15
 - radio groups 2-19
- ITypeComp interface 44-13
- ITypeInfo interface 44-13
- ITypeLib interface 44-13
- IUnknown interface 3-16, 3-20, 3-21, 44-3, 44-4, 44-16, 44-17, 47-7
 - implemented in
 - TInterfacedObject 3-17

J

- javascript libraries 14-30, 14-32
 - locating 14-31, 14-32
- joins 21-14
 - cached updates and 25-22
- just-in-time activation 14-6, 51-4

K

- K footnotes (Help systems) 38-5
- KeepConnection property 16-6, 17-7
- KeepConnections property 16-6
 - TSession component 13-4
- key fields 20-14
 - multiple 20-8, 20-12, 20-13
 - searching on alternative 20-8
- key violations 20-23
- keyboard events 26-5, 34-3, 34-9
 - internationalization 10-7
- keyboard mappings 10-8, 10-9
- keyboard shortcuts 2-14
 - adding to menus 5-19
 - key-down messages 42-8

- KeyDown method 42-9
- KeyExclusive property 20-7, 20-14
- KeyField property 26-14
- KeyFieldCount property 20-8
- keys
 - searching on 20-8
 - setting ranges 20-14
- KeyViolTableName property 20-24
- keywords 38-5
 - protected 34-5
- Kind property
 - bitmap buttons 2-15

L

- labels 2-20, 10-8, 26-2, 31-4
 - columns 26-18
- Last method 18-10
- late binding 14-24, 28-11
 - Automation 47-8
- Layout property 2-15
- LEpath compiler directive 9-12
- leap years 41-7
- Left property 2-11, 5-3
- LeftCol property 2-21
- Length function 3-29
- libraries
 - custom controls 31-4
- license agreement 11-11
- licensing
 - ActiveX controls 48-5, 48-7
- Licensing, ActiveX controls 48-5
- Licensing, for Internet Explorer 48-6
- lines
 - drawing 7-5, 7-9, 7-9 to 7-10, 7-26 to 7-27
 - changing pen width 7-5
 - event handlers 7-24
 - erasing 7-27
- Lines property 2-13, 33-8
- LineSize property 2-14
- LineTo method 7-4, 7-7, 7-9, 36-3
- Link HTML tag (<A>) 29-18
- links 20-25
- list boxes 2-17, 26-2, 26-11, 26-12, 41-1
 - data-aware 26-11
 - dragging items 6-2, 6-3
 - dropping items 6-3
 - owner-draw 6-11
 - draw-item events 6-15

- measure-item events 6-14
 - variant styles 6-12
 - storing properties
 - example 5-8
 - list controls 2-17 to 2-18
 - List property 16-16
 - listening connections 30-2, 30-3, 30-7, 30-9
 - closing 30-8
 - port numbers 30-5
 - ListField property 26-14
 - lists
 - string 2-28 to 2-32
 - using in threads 8-5
 - ListSource property 26-14
 - literals 19-21
 - live result sets 21-16, 21-16 to 21-17
 - restrictions 21-16, 21-17
 - updating 25-21
 - LNpath compiler
 - directive 9-12
 - Loaded method 33-10
 - LoadFromFile method 21-8, 23-16, 36-4
 - client datasets 13-17, 24-24
 - graphics 7-18
 - strings 2-28
 - LoadFromStream method
 - client datasets 24-24
 - loading
 - graphics 36-4
 - local databases 12-2
 - local networks 28-3
 - connecting 28-18 to 28-19
 - Local SQL 21-4, 21-14
 - local transactions 13-9
 - localaddr file 28-19
 - locales 10-1
 - data formats and 10-9
 - resource modules 10-9
 - localization 10-12
 - overview 10-1
 - localizing resources 10-9, 10-11, 10-12
 - Locate method 18-8, 18-14, 20-5
 - Lock method 8-6
 - locking objects
 - nesting calls 8-6
 - threads 8-6
 - LockList method 8-6
 - locks 20-5
 - logging in
 - databases 12-3
 - SQL servers 12-3
 - logical operators 18-17
 - logical values 26-2, 26-14
 - Login dialog box 17-6
 - login scripts 17-6
 - LoginPrompt property 17-6, 23-7
 - long strings 3-24
 - lookup combo boxes 26-2, 26-12 to 26-14
 - getting values 26-13, 26-14
 - setting properties 26-14
 - lookup fields 19-6, 26-13
 - caching values 19-10
 - defining 19-9, 26-20
 - naming 26-20
 - lookup list boxes 26-2, 26-12 to 26-14
 - getting values 26-13, 26-14
 - setting properties 26-14
 - Lookup method 18-8, 18-15, 20-5
 - lookup values 26-18
 - LookupCache property 19-10
 - LookupDataSet property 19-10, 19-13
 - LookupKeyFields
 - property 19-10, 19-13
 - LookupResultField
 - property 19-13
 - IParam parameter 37-2
 - LPK_TOOL.EXE 48-6
 - LUPackage compiler
 - directive 9-12
- ## M
-
- main form 5-1
 - main VCL thread 8-4
 - OnTerminate event 8-6
 - MainMenu component 5-16
 - maintained aggregates 12-13, 24-9 to 24-12
 - subtotals 24-11
 - MainWndProc method 37-3
 - Make licensed 48-5, 48-7
 - mapping data types 20-22
 - Mappings property 20-22
 - Margin property 2-15
 - marshaling 44-7
 - COM data 47-8
 - COM interfaces 44-7
 - CORBA interfaces 28-2
 - custom 28-12
 - dual interfaces 44-16
 - IDispatch interface 44-11
 - skeletons 28-7
 - mask edit controls 2-12
 - masks
 - editing data 19-15
 - master/detail forms 12-12, 20-24 to 20-26
 - cached updates and 25-6
 - master/detail
 - relationships 12-12
 - cascaded deletes 15-3
 - cascaded updates 15-3
 - client datasets 14-26, 24-3
 - multi-tiered
 - applications 14-26, 24-3
 - nested tables 14-26, 24-3
 - referential integrity 12-5
 - MasterFields property 20-24
 - MasterSource property 20-24
 - Max property
 - progress bars 2-21
 - track bars 2-14
 - MaxDimensions property 27-19
 - MaxLength property 2-12
 - memo fields 26-9
 - rich edit controls 26-10
 - MaxRecords property 14-33
 - MaxRows property 29-23
 - MaxSummaries property 27-19
 - MaxValue property 19-13
 - MBCS 3-26
 - MDI applications 4-1 to 4-2
 - creating 4-2
 - menus
 - merging 5-26 to 5-27
 - specifying active 5-27
 - media devices 7-30
 - media players 7-30 to 7-32
 - example 7-31
 - Memo control 2-12
 - memo controls 6-6
 - properties 2-13
 - memo fields 26-2, 26-9
 - rich edit 26-10
 - memory
 - conserving 32-8
 - decision components 27-19
 - freeing bitmap 7-20
 - leaks in forms 5-5
 - memory management
 - COM objects 3-16
 - components 2-9
 - interfaces 3-21
 - memos 33-8
 - menu bars
 - moving items 5-21
 - menu components 5-16

- Menu Designer 2-26
- Menu designer 5-15 to 5-17
 - context menu 5-22
- menu items 5-18 to 5-19
 - adding 5-18, 5-26
 - defined 5-15
 - deleting 5-19, 5-23
 - editing 5-22
 - grouping 5-19
 - moving 5-20
 - naming 5-17, 5-26
 - nesting 5-19
 - placeholders 5-23
 - separator bars 5-19
 - setting properties 5-22
 - underlining letters 5-19
- Menu property 5-27
- menus 5-15 to 5-26
 - accessing commands 5-19
 - adding 5-17 to 5-22
 - drop-down 5-19 to 5-20
 - from other
 - applications 5-27
 - adding images 5-21
 - disabling items 6-9
 - displaying 5-21, 5-23
 - handling events 2-26 to 2-27, 5-26
 - internationalizing 10-8, 10-9
 - moving among 5-23
 - naming 5-17
 - pop-up 6-10, 6-11
 - reusing 5-23
 - saving as templates 5-24, 5-25 to 5-26
 - templates 5-17, 5-23, 5-24 to 5-26
 - deleting 5-25
 - loading 5-24
- message handlers
 - declarations 37-4
- message headers (HTTP) 29-1, 29-2
- message loop
 - threads 8-4
- message-based servers
 - See also* Web server applications
 - threads 8-11
- messages 5-4, 37-1 to 37-7, 41-4
 - cracking 37-2
 - defined 37-2
 - dispatching 37-2
 - handlers 37-1, 37-2, 41-4
 - creating 37-5 to 37-7
 - declarations 37-5, 37-7
 - default 37-3
 - methods, redefining 37-6
 - overriding 37-3
 - handling 37-3 to 37-5
 - identifiers 37-5
 - key 42-8
 - mouse 42-8
 - records 37-2, 37-4
 - types, declaring 37-6
 - trapping 37-4
 - user-defined 37-5, 37-7
- metadata 20-22
 - obtaining from
 - providers 24-18
- metatables 2-22, 7-1, 7-16, 7-18, 36-4
 - when to use 7-3
- method pointers 34-2, 34-3, 34-8
- Method property 29-14
- methods 7-14, 31-6, 35-1, 41-10
 - ActiveX controls 48-9, 48-10
 - Automation and 47-7
 - calling 34-6, 35-3, 40-4
 - declaring 7-14, 35-4
 - dynamic 32-9
 - public 35-3
 - static 32-7
 - virtual 32-8
 - deleting 2-27
 - dispatching 32-7
 - drawing 40-8, 40-9
 - event handlers 34-3, 34-5
 - overriding 34-6
 - exposing for
 - Automation 47-3, 47-4
 - field objects 19-17
 - graphics 36-3, 36-4, 36-6, 36-7
 - palettes 36-5
 - inherited 34-6
 - initialization 33-10
 - message-handling 37-1, 37-3, 37-4
 - naming 35-2
 - objects and 2-2, 2-4
 - overriding 32-8, 37-3, 37-4, 41-11
 - properties and 33-5 to 33-6, 35-1, 35-2, 40-3
 - protected 35-3
 - public 35-3
 - redefining 32-8, 37-6
 - terminating 18-25
 - virtual 32-8, 35-4
- MethodType property 29-10, 29-14
- Microsoft Data Access SDK 23-4
- Microsoft IIS server
 - debugging 29-25
- Microsoft Server DLLs 29-5
 - creating 29-6
 - request messages 29-7
- Microsoft SQL Server 13-10, 13-12
- Microsoft SQL Server driver
 - deploying 11-5
- MIDAS 14-1, 14-2
 - deploying 11-5, 11-11
 - files 11-6
 - server licenses 14-2
 - Web applications 14-28
 - building 14-29 to 14-30, 14-30 to 14-38
- MIDAS page (Component palette) 2-10, 14-2, 14-19
- MIDAS.DLL 11-6, 12-15, 13-13, 14-2, 24-1
- MIDI files 7-31
- MIDL
 - creating type libraries 44-14
 - See also* IDL
- MIME messages 29-4
- Min property
 - progress bars 2-21
 - track bars 2-14
- MinSize property 2-15
- MinValue property 19-13
- MKTYPLIB 44-14
- MM film 7-31
- mobile computing 13-17
- modal forms 5-5
- Mode property 20-21
 - pens 7-5
- modeless forms 5-5, 5-6
- Modified method 42-11
- Modified property 2-13
- Modifiers property 2-14
- ModifyAlias method 16-11
- modifying records 20-21
- ModifySQL property 25-11
- modules 31-10
 - Type Library editor 50-21 to 50-23, 50-30
- Month property 41-5
- MonthCalendar
 - component 2-18
- months, returning current 41-7
- mouse buttons 7-23

- clicking 7-23, 7-24
 - mouse-move events
 - and 7-25
 - mouse events 7-22 to 7-25, 26-5, 40-2
 - defined 7-22
 - dragging and dropping 6-1 to 6-4
 - parameters 7-23
 - state information 7-23
 - testing for 7-25
 - mouse messages 37-2, 42-8
 - mouse pointer
 - drag-and-drop 6-4
 - MouseDown method 42-8
 - MouseToCell method 2-21
 - .MOV files 7-31
 - Move method
 - string lists 2-31, 2-32
 - MoveBy method 18-10
 - MoveCount property 20-23
 - MoveFile function 3-35
 - MovePt 7-27
 - MoveTo method 7-4, 7-7, 36-3
 - .MPG files 7-31
 - Msg parameter 37-3
 - MTS 4-11, 44-2, 51-1 to 51-23
 - activities 51-17
 - administering objects 51-22 to 51-23
 - base clients 51-2, 51-14
 - callbacks 51-19
 - components 51-2
 - database connections 14-6
 - debugging Web server applications 29-26 to 29-27
 - multi-tiered database applications 14-5 to 14-6
 - object contexts 51-6
 - object pooling 51-6
 - object references 51-18
 - packages 51-13
 - pooling database connections 14-6, 51-5
 - releasing resources 51-6
 - requirements 51-4
 - security 51-11
 - transactions 51-5, 51-7 to 51-11
 - attributes 51-7
 - client-controlled 51-10
 - timeouts 51-10
 - MTS Data Module wizard 14-14 to 14-15
 - MTS data modules
 - interface 14-17
 - threading models 14-14
 - transaction attributes 14-15
 - MTS Explorer 51-22 to 51-23
 - MTS Object wizard 51-15
 - MTS objects 51-2 to 51-4
 - administering 51-22 to 51-23
 - creating 51-15 to 51-17
 - debugging 51-20
 - installing 51-21
 - requirements 51-4
 - sharing properties 51-12
 - MTS packages 51-21
 - MTS proxy 14-18, 51-3
 - multidimensional
 - crosstabs 27-3
 - multi-homed hosts 28-19
 - multi-line text controls 26-9, 26-10
 - multimedia 7-31
 - multipage dialog boxes 2-19
 - multiple document interface 4-1 to 4-2
 - multiple forms 26-6, 26-7
 - multiprocessing
 - threads 8-1
 - multi-read exclusive-write synchronizer 8-7
 - warning about use 8-8
 - MultiSelect property 2-17
 - multi-table queries 21-14
 - multi-threaded
 - applications 8-1, 16-2, 16-16 to 16-17
 - distributed 8-11
 - Multi-tier Distributed Application Services Suite (MIDAS) 14-1, 14-2
 - Multitier page (New Items dialog) 14-3
 - multi-tiered applications 12-3, 12-6, 12-9, 14-1
 - advantages 14-2
 - architecture 14-4
 - building 14-10 to 14-25
 - callbacks 14-17
 - client-generated events 15-10, 24-23
 - data constraints 15-10
 - data providers 14-16, 15-1
 - defined 14-1
 - deploying 11-5
 - master/detail relationships 14-26
 - MIDAS Web
 - applications 14-28 to 14-38
 - overview 14-3 to 14-10
 - passing parameters 24-15 to 24-16
 - updating records 24-20 to 24-22
 - multi-tiered architecture 14-4
 - Web-based 14-28
 - mutually exclusive options 5-30
- ## N
-
- Name property 19-13, 26-6
 - menu items 2-26, 2-27
 - naming
 - database sessions 29-22
 - naming conventions
 - events 34-8
 - fields 34-2
 - message-record types 37-6
 - methods 35-2
 - properties 33-6
 - navigating datasets 18-9 to 18-12, 18-19
 - navigator 18-9, 18-10, 26-2, 26-29 to 26-31
 - buttons 26-29
 - deleting data 18-22
 - editing and 18-21
 - enabling/disabling buttons 26-30
 - help hints 26-31
 - nested details 19-26 to 19-27, 20-26
 - fetch on demand 15-3, 24-17
 - nested tables 12-13, 19-26 to 19-27, 20-26
 - client datasets 14-26
 - flat files 24-3
 - master/detail relationships 14-26
 - NetBEUI protocol 17-7
 - NetFileDir property 16-13
 - Netscape Server DLLs 29-5
 - creating 29-6
 - request messages 29-7
 - Netscape servers
 - debugging 29-27
 - network control files 16-13
 - networks
 - accessing data 25-1
 - communication layer 28-2
 - connecting to 17-7
 - temporary tables and 16-13
 - New command 31-10

- New Field dialog box 19-7
 - defining fields 19-8, 19-10, 19-11
 - New Items dialog 2-35, 2-36, 2-37
 - New Thread Object dialog 8-2
 - newsgroups 1-3
 - NewValue property 25-26
 - Next method 18-10
 - non-blocking connections 30-10
 - to 30-11
 - blocking vs. 30-10
 - nonindexed fields
 - searching on 20-6
 - no-nonsense license agreement 11-11
 - non-production index files 20-9
 - nonvisual components 19-2, 26-5, 31-4, 31-11, 43-2
 - nonvisual objects 2-8
 - NOT NULL constraint 15-10
 - NOT NULL UNIQUE constraint 15-10
 - notebook dividers 2-19
 - notification events 34-7
 - NSAPI applications 4-10, 29-5
 - creating 29-6
 - debugging 29-25
 - request messages 29-7
 - null values 18-24
 - ranges and 20-13
 - null-terminated
 - wide strings 3-25
 - numbers 33-2
 - formatting 19-16
 - internationalizing 10-9
 - property values 33-9
 - numeric fields 19-16
 - NumGlyphs property 2-15
- ## O
-
- OAD 28-3 to 28-4, 28-8
 - registering servers 28-9 to 28-11
 - running 28-9
 - unregistering objects 28-10
 - oadutil 28-10
 - Object Activation Daemon (OAD) 28-3 to 28-4, 28-8
 - registering servers 28-9 to 28-11
 - running 28-9
 - unregistering objects 28-10
 - Object Broker 14-22
 - object contexts 51-6
 - transactions 51-8
 - object fields 19-23 to 19-28
 - types 19-23
 - Object HTML tag (<OBJECT>) 29-18
 - Object Inspector 2-4, 2-23, 33-2, 38-6
 - editing array properties 33-2
 - help with 38-4
 - selecting menus 5-24
 - Object Management Group (OMG) 28-1
 - Object Pascal
 - overview 2-1
 - object pooling 51-6
 - remote data modules 14-7
 - Object Repository 2-35 to 2-37, 5-11
 - adding items 2-35
 - converting Web server applications 29-28
 - specifying shared directory 2-35
 - using items from 2-36
 - Object Request Broker (ORB) 28-1, 28-14
 - object variables 2-7
 - ObjectContext
 - example 51-20
 - ObjectName property
 - TCorbaConnection 14-22
 - object-oriented
 - programming 2-2 to 2-9, 32-1 to 32-9
 - declarations 32-3, 32-9
 - classes 32-5, 32-6
 - methods 32-7, 32-8, 32-9
 - defined 2-2
 - distributed
 - applications 28-1, 28-2
 - inheritance 2-5
 - objects 2-2 to 2-9
 - accessing 2-6 to 2-7
 - creating 2-8
 - customizing 2-5
 - defined 2-2
 - destroying 2-9
 - distributed 28-1
 - dragging and dropping 6-1
 - events and 2-4
 - helper 2-27
 - inheritance 2-5 to 2-6
 - instantiating 2-3, 34-2
 - multiple instances 2-3
 - nonvisual 2-8
 - owned 40-5 to 40-7
 - initializing 40-6
 - properties 2-2
 - stateless 51-8
 - temporary 36-6
 - type declarations 2-7
 - Objects property 2-22
 - string lists 2-32, 6-14
 - .OCX files 11-3
 - ODBC drivers 17-7, 17-8
 - isolation levels 13-8
 - using with ADO 13-11, 13-12, 23-2
 - ODL (Object Description Language) 44-12, 44-14, 50-1
 - OEM character sets 10-2
 - OEMConvert property 2-13
 - offscreen bitmaps 36-5 to 36-7
 - OldValue property 25-10, 25-26
 - OLE
 - connecting to application servers 14-21
 - merging menus 5-26
 - OLE Automation *see* Automation
 - OLE DB 13-11, 13-12, 23-2
 - OLE32.dll 44-2
 - OLEAut32.dll 44-2
 - OLEEnterprise 11-5, 14-10, 14-12
 - connecting to application servers 14-21
 - OLEView 44-14
 - OMG 28-1
 - OnAccept event
 - server sockets 30-9
 - OnAction event 29-12
 - OnAfterPivot event 27-9
 - OnBeforePivot event 27-9
 - OnBeginTransComplete event 23-11
 - OnCalcFields event 18-8, 18-26, 19-8, 19-9, 24-9
 - OnCellClick event 26-27
 - OnChange event 19-16, 36-7, 40-7, 41-11, 42-10
 - OnClick event 2-15, 34-1, 34-2, 34-5
 - buttons 2-3
 - menus 2-26
 - OnClientConnect event 30-7
 - server sockets 30-10
 - OnClientDisconnect event 30-8
 - OnClientRead event
 - server sockets 30-10
 - OnClientWrite event

- server sockets 30-10
- OnColEnter event 26-27
- OnColExit event 26-27
- OnColumnMoved event 26-26, 26-27
- OnCommitTransComplete event 23-11
- OnConnect event
 - client sockets 30-9
- OnConnectComplete event 23-4
- OnConnecting event
 - client sockets 30-9
- OnConstrainedResize event 5-4
- OnCreate event 31-12
- OnDataChange event 26-7, 42-6, 42-10
- OnDataRequest event 15-10, 24-23
- OnDbClick event 26-27, 34-5
- OnDecisionDrawCell event 27-12
- OnDecisionExamineCell event 27-13
- OnDisconnect event 23-5
 - client sockets 30-6
- OnDragDrop event 6-2, 26-27, 34-5
- OnDragOver event 6-2, 26-27, 34-5
- OnDrawCell event 2-21
- OnDrawColumnCell event 26-26, 26-27
- OnDrawDataCell event 26-27
- OnDrawItem event 6-15
- OnEditButtonClick event 26-27
- OnEndDrag event 6-3, 26-27, 34-5
- OnEndPage 49-3
- OnEnter event 26-27, 26-30, 26-31, 34-5
- OnError event
 - sockets 30-8
- one-to-many relationships 20-24
- OnExit event 26-27
- OnFilterRecord event 18-8, 18-16, 18-18
- OnGetData event 15-5
- OnGetDataSetProperties event 15-4
- OnGetSocket event
 - server sockets 30-9
- OnGetTableName event 15-10
- OnGetText event 19-16, 19-17
- OnGetThread event 30-10
- OnHTMLTag event 14-38, 29-19, 29-20, 29-21
- OnKeyDown event 26-27, 34-5, 42-9
- OnKeyPress event 26-27, 34-5
- OnKeyUp event 26-27, 34-5
- OnLayoutChange event 27-9
- online help 38-4
- OnListen event
 - server sockets 30-9
- OnLogin event 17-2, 17-6, 23-7
- OnLookup event
 - client sockets 30-9
- OnMeasureItem event 6-14
- OnMouseDown event 7-23, 34-5, 42-8
 - parameters passed to 7-23
- OnMouseMove event 7-23, 7-24, 34-5
 - parameters passed to 7-23
- OnMouseUp event 7-13, 7-23, 7-24, 34-5
 - parameters passed to 7-23
- OnNewDimensions event 27-9
- OnPaint event 2-22, 7-2
- OnPassword event 16-15, 17-2
- OnPopup event 6-11
- OnRead event
 - client sockets 30-10
- OnReconcileError event 24-21
- OnRefresh event 27-7
- OnResize event 7-2
- OnRollbackTransComplete event 23-11
- OnScroll event 2-13
- OnSetText event 19-16, 19-17
- OnStartDrag event 26-27
- OnStartPage 49-3
- OnStartup event 16-5
- OnStateChange event 18-5, 26-7
- OnSummaryChange event 27-9
- OnTerminate event 8-6
- OnThreadStart event
 - server sockets 30-10
- OnTitleClick event 26-27
- OnUpdateData event 15-6, 26-7
- OnUpdateError event 15-9, 18-29, 25-9, 25-23
- OnUpdateRecord event 18-29, 25-24, 25-25
 - cached updates 25-22
 - update objects 25-11, 25-18, 25-20, 25-22
- OnValidate event 19-16
- OnWillConnect event 23-4
- OnWrite event
 - client sockets 30-10
- Open method 23-23
 - ADO connection components 23-4
 - databases 17-7
 - datasets 18-3
 - queries 21-12, 21-13
 - server sockets 30-7
 - sessions 16-5
 - tables 20-4
- OpenDatabase method 16-5, 16-6
- OpenSession method 16-16
- OpenString 3-25
- operators
 - data filters 18-17
 - optimizing code 7-14
 - interfaces 3-21
 - optimizing system resources 31-4
 - optional parameters 15-4, 24-12
- Options property 2-21
 - data grids 26-24, 26-25
 - decision grids 27-12
 - providers 15-3
- Oracle 13-12
- Oracle drivers
 - deploying 11-5
- Oracle tables 22-17
- Oracle8 tables 12-13
 - limits on creating 13-10
- ORB 28-1, 28-14
- ORB domains 28-18
- ORDER BY clause 20-10
- Orientation property
 - data grids 26-28
 - track bars 2-14
- Origin property 7-26, 19-13
- osagent 28-2, 28-3, 28-17
- outlines
 - drawing 7-5
- out-of-process servers 44-6
 - for Active Server Pages 49-3
- output parameters 22-10
- Overload property 22-17
- overloaded stored procedures 22-17
- override directive 32-8
- overriding
 - methods 32-8, 37-3, 37-4, 41-11
- owned objects 40-5 to 40-7
 - initializing 40-6
- Owner property 2-9, 31-12

- owner-draw controls 2-32, 6-11
 - drawing 6-13, 6-14
 - list boxes 2-17, 2-18
 - sizing 6-13
 - styles 6-12

P

- \$P compiler directive 3-32
- Package Collection Editor 9-14
- package collection files 9-13
- package files 11-3
- packages 9-1 to 9-15, 38-19
 - collections 9-13
 - compiler directives 9-11
 - compiling 9-10 to 9-13
 - options 9-11
 - components 38-19
 - Contains list 9-7, 9-10, 38-19
 - creating 4-8, 9-7 to 9-12
 - custom 9-5
 - default settings 9-8
 - deploying applications 9-2, 9-13
 - design-only option 9-8
 - design-time 9-1, 9-5 to 9-7
 - DLLs 9-1, 9-2
 - duplicate references 9-10
 - editing 9-8
 - file-name extensions 9-1
 - installing 9-6 to 9-7
 - internationalizing 10-11, 10-12
 - options 9-8
 - referencing 9-3
 - Requires list 9-7, 9-8, 9-9, 38-19
 - runtime 9-1, 9-2 to 9-5, 9-8
 - source files 9-2, 9-13
 - using 4-8
 - using in applications 9-3 to 9-5
- Packages page (Web deployment) 48-22
- page controls 2-19
- page producers 29-17 to 29-21
 - chaining 29-20
 - converting templates 29-19
 - data-aware 14-35 to 14-38, 29-22
 - event handling 29-19, 29-20, 29-21
- PageSize property 2-14
- paint boxes 2-22
- Paint method 36-6, 40-8, 40-9
- palette bitmap files 38-3

- PaletteChanged method 36-5
- palettes 36-4 to 36-5
 - default behavior 36-5
 - specifying 36-5
- PanelHeight property 26-28
- panels 2-19
 - adding speed buttons 5-29
 - attaching to form tops 5-29
 - beveled 2-22
 - speed buttons 2-15
- PansiChar 3-25
- PAnsiString 3-30
- Paradox tables 16-2, 20-2, 20-3
 - accessing data 20-5, 21-4
 - adding records 18-22
 - batch moves 20-23, 20-24
 - creating aliases 16-11
 - DatabaseName 13-3
 - directories 13-4
 - insufficient access
 - rights 16-15
 - isolation levels 13-8
 - local transactions 13-9
 - memo fields 26-9, 26-10
 - network control files 16-13
 - opening connections 16-6
 - password protecting 16-13
 - queries 21-16
 - retrieving indexes 20-9
 - searching 20-6, 20-8
 - temporary files 16-13
- parallel processes
 - threads 8-1
- ParamBindMode
 - property 22-15
- ParamByName method 21-10, 23-17, 23-29
- parameter substitution (SQL) 25-14, 25-18
- parameterized queries 21-6
 - creating 21-8 to 21-11
 - at runtime 21-10
 - defined 21-2
 - running from text files 21-8
- parameters
 - classes as 32-9
 - client datasets 24-15 to 24-16
 - DII calls 28-14
 - dual interfaces 47-9
 - event handlers 34-3, 34-7, 34-8, 34-9

- HTML tags 29-18
- messages 37-2, 37-3, 37-4, 37-6
- mouse events 7-23
- multi-tiered
 - applications 24-16
 - property settings 33-6
 - array properties 33-8
- Parameters property 23-17, 23-28
- ParamName property 14-36
- Params property 17-5, 17-6
 - queries 21-10
 - XML brokers 14-33
- parent controls 2-11
- parent properties 2-11
- Parent property 31-12
- ParentShowHint property 2-21
- partial keys
 - searching on 20-8
 - setting ranges 20-14
- passthrough SQL 13-6, 13-8, 21-17
- PasswordChar property 2-13
- passwords 17-2
 - Paradox tables 16-13
 - sessions and 13-4
- PasteFromClipboard
 - method 6-9, 26-10
 - graphics 26-10
- PathInfo property 29-10
- paths (URLs) 29-2
- patterns 7-8
- pbmByName const 22-15
- pbmByNumber const 22-15
- .PCE files 9-13
- PChar 3-25
 - string conversions 3-30
- PDOXUSRS.NET 16-13
- Pen property 7-3, 7-5, 36-3
- PenPos property 7-3, 7-7
- pens 7-5, 40-5
 - brushes 7-5
 - changing 40-7
 - colors 7-5
 - default settings 7-5
 - drawing modes 7-27
 - getting position of 7-7
 - position, setting 7-7, 7-24
 - style 7-6
 - width 7-5
- PENWIN.DLL 9-12
- performance, optimizing in type libraries 50-11
- persistent columns 26-16, 26-18

- adding buttons 26-21
- creating 26-19 to 26-22
- deleting 26-17, 26-19, 26-20
- inserting 26-19
- reordering 26-20, 26-26
- persistent connections 16-6, 16-7
- persistent field lists 19-4
 - viewing 19-5, 19-6
- persistent fields 19-4, 26-16
 - creating 19-5, 19-6
 - creating client datasets 13-14
 - data packets and 15-2
 - deleting 19-11
 - ordering 19-6
- Personal Web server
 - debugging 29-27
- pick lists 26-20
- PickList property 26-20, 26-22
- picture objects 7-3, 36-4
- Picture property 2-22, 7-16
 - in frames 5-14
- pictures 7-16, 36-3 to 36-5
 - changing 7-19
 - loading 7-18
 - replacing 7-19
 - saving 7-19
- Pie method 7-4
- Pixel property 7-3, 36-3
- pixels
 - reading and setting 7-9
- Pixels property 7-5, 7-9
- pmCopy constant 7-27
- pmNotXor constant 7-27
- pointers
 - Automation 47-7
 - class 32-9
 - default property values 33-9
 - method 34-2, 34-3, 34-8
- Polygon method 7-4, 7-11
- polygons 7-11
 - drawing 7-11
- PolyLine method 7-4, 7-10
- polylines 7-9, 7-10
 - drawing 7-9
- polymorphism 2-2
- pop-up menus 6-10 to 6-11
 - displaying 5-21
 - drop-down menus and 5-20
- PopupMenu component 5-16
- PopupMenu property 6-10
- Port property
 - client sockets 30-6
 - server sockets 30-7
 - TSocketConnection 14-20
- ports 30-5
 - client sockets 30-6
 - multiple connections 30-5
 - ORB domains 28-18
 - server sockets 30-7
 - services and 30-2
- Position property 2-14, 2-21
- Post method 18-6, 18-7, 18-23
 - Edit and 18-21
- posting records 18-23, 26-4
 - data grids 26-25
- Precision property 19-13
- preexisting controls 31-4
- Prepare method 21-7, 21-13, 22-5
- preparing queries 21-14
- primary indexes 20-9
 - batch moves and 20-21, 20-22
- PRIMARY KEY constraint 15-10
- principals 28-16
- Prior method 18-10
- priorities
 - using threads 8-1, 8-2
- Priority property 8-3
- private 2-7
- private directories 16-13
- private properties 33-5
- PrivateDir property 16-13
- privileges 20-4
- problem tables 20-23
- ProblemCount property 20-23
- ProblemTableName
 - property 20-23, 20-24
- ProcedureName property 23-22
- procedures 31-6, 34-3
 - naming 35-2
 - property settings 38-11
- progress bars 2-21
- Project Manager 5-2
- project options 4-2
 - default 4-2
- Project Options dialog box 4-2
- Project page (Web deployment) 48-20
- project templates 2-36
- projects
 - adding forms 5-1 to 5-2
 - properties 33-1 to 33-11
 - accessing 33-5 to 33-6
 - ActiveX controls 48-9, 48-13
 - array 33-2, 33-8
 - as classes 33-2
 - Automation 47-7
 - BDE-enabled datasets 18-28
- changing 38-6 to 38-12, 39-2, 39-3
- columns 26-17, 26-18, 26-21
 - resetting 26-22
- common dialog boxes 43-1
- data sources 26-6 to 26-7
- decision cubes 27-7
- decision grids 27-12
- decision pivots 27-10
- decision sources 27-9
- declaring 33-2, 33-3, 33-3 to 33-6, 33-7, 33-10, 34-8, 40-3
 - stored 33-10
 - user-defined types 40-3
- default values 33-7, 33-9 to 33-10
 - redefining 39-2, 39-3
- editing
 - as text 38-8
- events and 34-1, 34-2
- exposing for
 - Automation 47-3
- field objects 19-3, 19-12 to 19-15
- grids 26-17, 26-28
- help with 38-4
- HTML tables 29-23
- inherited 33-2, 40-2, 41-2
- internal data storage 33-4, 33-6
- loading 33-10
- lookup combo boxes 26-14
- lookup list boxes 26-14
- nodefault 33-7
- objects and 2-2
- overview 31-6
- published 41-2
- read and write 33-5
- read-only 32-6, 33-6, 42-2
- redeclaring 33-9, 34-5
- rich text controls 2-13
- setting 2-23 to 2-24
- sharing among MTS
 - objects 51-12
- specifying values 33-9, 38-8
- storing 33-10
- storing and loading
 - unpublished 33-11 to 33-13
- types 33-2, 33-8, 38-8, 40-3
- updating 31-7
- viewing 38-8
- wrapper components 43-3
- write-only 33-6

property editors 2-23, 33-2, 38-6 to 38-12

- as derived classes 38-7
- attributes 38-10
- dialog boxes as 38-9
- registering 38-11 to 38-12
- property pages
 - ActiveX controls 48-12, 48-15
 - adding controls to 48-14
 - associating with ActiveX control properties 48-14
 - creating 48-14
 - updating 48-14
 - updating ActiveX controls 48-15
- property settings
 - reading 33-8
 - writing 33-8
- protected 2-7
 - directive 34-5
 - events 34-5
 - keyword 33-3, 34-5
 - part of classes 32-5
- protocols
 - choosing 14-8 to 14-10
 - connection
 - components 14-19
 - Internet 29-1, 30-1
 - network connections 17-7
- ProviderFlags property 15-8
- ProviderName property 14-18, 14-33
- providers 12-10, 14-3, 14-16, 15-1 to 15-11
 - associating with datasets 15-1
 - client datasets and 24-14 to 24-23
 - controlling data packets 15-1
 - data constraints 15-10
 - error handling 15-9
- proxy 44-7
- PString 3-30
- public 2-7
 - directive 34-5
 - keyword 34-5
 - part of classes 32-6
 - properties 33-9
- published 2-7, 33-2
 - directive 33-2, 34-5, 43-3
 - keyword 34-5
 - part of classes 32-6
 - properties 33-9, 33-10
 - example 40-2, 41-2
- PWideChar 3-25
- PWideString 3-30

Q

- QReport page (Component palette) 2-10
- qualifiers 2-6 to 2-7
- queries 12-13, 18-3, 21-1
 - ADO-based 12-14
 - cached updates and 25-22
 - creating 21-4, 21-6
 - at runtime 21-7
 - defining statements 21-5 to 21-8
 - filtering vs. 18-16
 - heterogenous 21-14
 - HTML tables 29-24
 - Interbase 12-14
 - multi-table 21-14
 - named sessions and 16-2
 - optimizing 21-12, 21-15
 - overview 21-1 to 21-5
 - parameter
 - substitution 25-14, 25-18
 - preparing 21-13, 21-14
 - result sets 21-13, 21-16 to 21-17
 - cursors and 21-15
 - getting at runtime 21-12
 - updating 21-17, 25-21
 - running 21-12 to 21-13, 25-19
 - from text files 21-8
 - setting parameters 21-8 to 21-11
 - at runtime 21-10
 - special characters and 21-6
 - submitting statements 21-13
 - update objects and 25-11, 25-13
 - Web applications 29-24
 - whitespace characters and 21-6
- Query Builder 21-7
- query components 12-13, 21-1
 - adding 21-4
- Query Parameters editor 21-9
- query part (URLs) 29-2
- Query property
 - update objects 25-16
- QueryInterface method 3-16, 3-20, 3-21
 - IUnknown 44-4

R

- radio buttons 2-16, 26-2
 - data-aware 26-15
 - grouping 2-19

- selecting 26-15
- radio groups 2-19
- raise reserved word 3-13
- ranges 20-11 to 20-16
 - applying 20-14
 - boundaries 20-14
 - changing 20-15
 - filters vs. 20-11
 - specifying 20-13
- raster operations 36-7
- .RC files 5-27
- RDBMS 12-2, 14-1
- Read method
 - TFileStream 3-38
- read method 33-6
- read reserved word 33-8, 40-4
- ReadBuffer method
 - TFileStream 3-38
- reading property settings 33-6, 38-8
- README.TXT 11-10, 11-11
- read-only data
 - client datasets 24-4
- read-only fields 26-3
- read-only properties 32-6, 33-6, 42-2
- ReadOnly property 2-12, 42-3, 42-9
 - data grids 26-22, 26-25
 - data-aware controls 19-13, 26-3
 - memo fields 26-9
 - rich edit controls 26-10
 - tables 20-5
- read-only records 18-7
- read-only result sets 21-17
 - updating 25-21
- read-only tables 20-5
- realizing palettes 36-5
- ReasonString property 29-16
- rebars 5-28, 5-33
- RecNo property
 - client datasets 24-2
- reconciling data 24-21
- RecordCount property 20-23
- records
 - adding 18-7, 18-21 to 18-22, 18-24
 - appending 18-22
 - batch operations 20-19, 20-21
 - cached updates and 25-3
 - changing 18-23
 - comparison to objects 2-2
 - deleting 18-22, 20-16

- batch operations 20-22
- caution 20-16
- displaying 26-27
 - refresh intervals 26-4, 26-5
- fetching 25-3
- filtering 18-8, 18-16 to 18-19
- finding 18-8, 18-14 to 18-16, 20-5 to 20-8
 - specific ranges 20-12 to 20-16
- getting subsets 20-11 to 20-16
- iterating through 18-11
- marking 18-12 to 18-14
- moving through 18-9 to 18-12, 18-19, 26-7, 26-29
- posting 18-23, 26-4
 - data grids 26-25
- read-only 18-7
- reconciling updates 24-21
- refreshing 24-22 to 24-23
- repeating searches 20-8
- setting current 20-7
- sorting 20-9 to 20-11
 - with alternative indexes 20-9
- synchronizing current 20-24
- tracking moves 20-23
- Type Library editor 50-19 to 50-20, 50-29
- undeleting 25-8 to 25-10
- updating 18-24, 26-7
 - batch operations 20-21
 - multiple datasets 25-5
 - multi-tiered clients 24-20 to 24-22
 - queries and 21-17
- RecordSet property 23-14, 23-28
- RecordSetState property 23-14
- Rectangle method 7-4, 7-10, 36-3
- rectangles
 - drawing 7-10, 40-9
- redefining methods 32-8
- redrawing images 36-7
- reference counting
 - COM interfaces 44-4
 - COM objects 3-16, 44-4
 - interfaces 3-20 to 3-22
- reference fields 19-27 to 19-28
- references
 - forms 5-2
 - packages 9-3
- referential integrity 12-5
- Refresh 5-41
- refresh intervals 26-4, 26-5
- Refresh method 26-5
- RefreshLookupList
 - property 19-11
- RefreshRecord method 24-22
- Register method 7-3
- Register procedure 31-11, 38-2
- RegisterComponents
 - procedure 9-7, 31-11, 38-2
- registering
 - ActiveX controls 48-17
 - component editors 38-15
 - components 31-11
 - CORBA interfaces 28-8
 - property editors 38-11 to 38-12
- registering, Active Server Pages 49-4
- RegisterPooled procedure 14-7
- RegisterPropertyEditor
 - procedure 38-11
- RegisterTypeLib function
 - type libraries 44-14
- Registry 10-9
- registry 2-32
- REGSERV32.EXE 11-3, 11-6
- relational databases 12-1, 15-10, 21-1
- Release method 3-16, 3-20, 3-21
 - IUnknown 44-4
 - TCriticalSection 8-7
- release notes 11-11
- releasing mouse buttons 7-24
- remote applications
 - batch operations 20-22
 - cached updates and 25-1
 - retrieving data 21-1, 21-3, 21-14, 21-17
 - TCP/IP 30-1
- remote connections 17-7 to 17-8, 18-28, 30-2 to 30-3
 - changing 14-23
 - dropping 14-23
 - multiple 30-5
 - opening 14-19, 30-6, 30-7
 - sending/receiving information 30-10
 - terminating 30-8
 - unauthorized access 17-6
- Remote Data Module wizard 14-13 to 14-14
- remote data modules 2-34, 14-3, 14-5, 14-11, 14-13 to 14-16
 - instanting 14-14
- interfaces 14-16 to 14-18
 - pooling 14-7
 - stateless 14-6, 14-7, 14-26 to 14-27
 - threading models 14-13, 14-14
- remote database management systems 12-2
- remote database servers *See* remote servers
- Remote DataBroker 11-5
- remote servers 12-2, 16-6, 16-11, 21-4, 44-6
 - accessing data 25-1
 - attaching 17-7 to 17-8
 - names 14-19
 - unauthorized access 17-6
- RemoteServer property 14-18, 14-33
- RemovePassword
 - method 16-14
- RenameFile function 3-35, 3-36
- repainting controls 40-7, 40-8, 41-4
- reports 12-15
- Repository 2-35 to 2-37, 5-11
 - adding items 2-35
 - using items from 2-36
- Repository dialog 2-35
- Repository IDs 28-10
- RepositoryID property 14-22
- Request for Comment (RFC) documents 29-1
- request headers 29-13
- request messages 29-7
 - action items and 29-10
 - contents 29-15
 - dispatching 29-9
 - header information 29-13 to 29-15
 - HTTP overview 29-3 to 29-4
 - processing 29-9
 - responding to 29-12 to 29-13, 29-16
 - types 29-14
- request objects
 - header information 29-8
- RequestLive property 21-16, 25-21
- Requires list (packages) 9-7, 9-8, 9-9, 38-19
- ResetEvent method 8-9
- resizing controls 2-14, 11-8, 41-4
 - graphics 36-7
- resolvers 15-1, 15-6

- resorting fields 20-10
- resource dispensers 51-5, 51-11 to 51-13
- Resource DLLs
 - dynamic switching 10-12
 - wizard 10-9
- resource files 5-27 to 5-28
 - and type libraries 44-13
 - loading 5-27
- resource modules 10-9, 10-11
- resource pooling 51-5 to 51-6
- resources 31-7, 36-1
 - caching 36-2
 - freeing 43-5
 - isolating 10-9
 - localizing 10-9, 10-11, 10-12
 - strings 10-9
 - system, optimizing 31-4
- resourcestring reserved word 10-9
- response headers 29-16
- response messages 29-7
 - contents 29-16, 29-17 to 29-24
 - creating 29-15 to 29-17, 29-17 to 29-24
 - database information 29-21 to 29-24
 - header information 29-15 to 29-16
 - sending 29-12, 29-17
 - status information 29-15
- response templates 29-17
- RestoreDefaults method 26-19
- restoring deleted records 25-8
- Result data packet 24-20
- Result parameter 22-10, 37-6
- result sets 21-13, 21-16 to 21-17
 - cursors and 21-15
 - editing 21-16
 - getting at runtime 21-12
 - read-only 25-21
 - updating 21-17, 25-21
- Resume method 8-10
- ReturnValue property 8-8
- reusing code
 - techniques 5-11
- RevertRecord method 18-29, 25-8, 25-9
- RFC documents 29-1
- rich edit controls 6-6
- rich text controls 2-13
 - properties 2-13
- rich text edit controls 2-12, 26-10
 - properties 2-13
- role-based security 51-11
- Rollback method 13-7
- RollbackTrans method 23-11
- rolling back transactions 13-7
- rounded rectangles 7-11
- RoundRect method 7-4, 7-11
- RowAttributes property 29-23
- RowCount property 26-14, 26-29
- RowHeights property 2-21, 6-14
- rows 2-21, 18-21
 - decision grids 27-11
- Rows property 2-22
- RPC 44-8
- rtDeleted constant 25-9
- rtInserted constant 25-9
- rtModified constant 25-9
- RTTI 32-6
- rtUnmodified constant 25-9
- rubber banding example 7-22 to 7-27
- running queries 21-12 to 21-13
 - from text files 21-8
 - update objects 25-19
- running stored procedures 22-5
- \$RUNONLY compiler
 - directive 9-11
- runtime interfaces 32-6
- runtime license 48-5, 48-7
- runtime packages 9-1, 9-2 to 9-5
- runtime type information 32-6

S

- safe arrays 50-25
- safecall calling
 - convention 48-10, 50-10
- SafeRef 51-18
- Samples page (Component palette) 2-10
- Save as Template command (Menu designer) 5-23, 5-25
- Save Attributes command 19-14
- Save Template dialog box 5-25
- Save To File command 13-15
- SaveConfigFile method 16-10
- SaveToFile method 7-19, 23-16, 36-4
 - client datasets 13-17, 24-25
 - strings 2-28
- SaveToStream method
 - client datasets 24-25
- saving
 - graphics 36-4
- scalability 12-6, 13-18
- ScaleBy property
 - TCustomForm 11-8
- Scaled property
 - TCustomForm 11-8
- ScanLine property
 - bitmap 7-9
 - bitmap example 7-17
- ScktSrvr.exe 14-9, 14-12, 14-20
- SCM 4-3
- scope (objects) 2-6 to 2-7
- screen
 - refreshing 7-2
 - resolution 11-7
 - programming for 11-8
- Screen variable 5-3, 10-8
- scripts (URLs) 29-2
- scroll bars 2-13
 - text windows 6-7
- scroll boxes 2-19
- scrollable bitmaps 7-16
- scrollable lists 26-11
- ScrollBars property 2-21, 6-7
 - memo fields 26-9
- SDI applications 4-1 to 4-2
- search criteria 18-14, 18-15
- search lists (Help systems) 38-5
- searching for data
 - incremental searches 26-14
- secondary indexes
 - searching with 20-8
- Sections property 2-20
- security 17-6
 - databases 12-3
 - dBase tables 13-4
 - DCOM 14-32
 - MTS 14-5, 14-8, 51-11
 - multi-tiered
 - applications 14-2
 - Paradox tables 13-4
 - registering for socket connections 14-9
 - scalability and 12-7
 - Web connections 14-9, 14-21
- seeking
 - files 3-39
- Select Menu command (Menu designer) 5-23
- Select Menu dialog box 5-23
- SELECT statements 21-13, 21-16
- SelectAll method 2-13
- SelectCell method 41-12, 42-3
- Selection property 2-21
- SelEnd property 2-14
- Self parameter 31-12
- SelLength property 2-13, 6-8
- SelStart property 2-13, 2-14, 6-8

- SelText property 2-13, 6-8
- Sender parameter 2-26
 - example 7-6
- separator bars (menus) 5-19
- Server 49-4
- server applications
 - architecture 14-4
 - automation interfaces 47-7
 - CORBA 28-2, 28-4 to 28-11
 - data constraints 15-10
 - interfaces 30-2
 - multi-tiered 14-4, 14-11 to 14-18
 - data providers 14-16, 15-1
 - overview 14-1
 - registering 14-11, 14-12, 28-8 to 28-11
 - OAD 28-4
 - retrieving data 21-3, 21-14, 21-17
 - services 30-1
 - threads 8-11
 - transactions 13-8
- server connections 30-2, 30-3
 - port numbers 30-5
- server sockets 30-6 to 30-8
 - accepting client requests 30-6
 - accepting clients 30-9
 - error messages 30-8
 - event handling 30-9
 - specifying 30-5, 30-6
 - using threads 30-11
 - Windows socket objects 30-7
- ServerType property 30-10, 30-11
- service applications 4-3 to 4-7
 - example code 4-4, 4-6
- Service Control Manager 4-3
- Service property
 - client sockets 30-6
 - server sockets 30-7
- Service Start name 4-7
- service threads 4-5
- services 4-3 to 4-7
 - CORBA 28-1
 - directory 28-2, 28-3
 - example code 4-4, 4-6
 - implementing 30-1 to 30-2, 30-6, 30-7
 - installing 4-3
 - name properties 4-7
 - network servers 30-1
 - ports and 30-2
 - requesting 30-5
 - uninstalling 4-4
- Session component 16-1, 16-2
- Session property 17-4
- SessionName property 16-4, 17-4, 18-28, 29-22
- sessions 13-4, 16-1, 17-9
 - activating 16-4, 16-5
 - alias names and 16-10
 - configuration modes 16-10
 - counting 16-16
 - creating 16-3, 16-16, 16-17
 - current state 16-4
 - default 16-2
 - disabling connections 16-5
 - getting information 16-8
 - managing aliases 13-4
 - multiple 16-1, 16-3, 16-16
 - naming 16-4, 29-22
 - restarting 16-5
 - testing associations 16-8, 16-12
 - Web applications 29-21
- Sessions property 16-16
- set types 33-2
- SetAbort 51-5, 51-6, 51-9
- SetBrushStyle method 7-8
- SetComplete 51-5, 51-6, 51-9
- SetComplete method
 - MTS data modules 14-17
- SetData method 19-17
- SetEvent method 8-9
- SetFields method 18-24
- SetFloatValue method 38-8
- SetKey method 18-8, 20-6
 - EditKey vs. 20-8
- SetLength procedure 3-29
- SetMethodValue method 38-8
- SetOrdValue method 38-8
- SetParams method 25-18
- SetPenStyle method 7-6
- SetRange method 20-13, 20-14
- SetRangeEnd method 20-13
 - SetRange vs. 20-13
- SetRangeStart method 20-12
 - SetRange vs. 20-13
- sets 33-2
- SetStrValue method 38-8
- SetValue method 38-8
- Shape property 2-22
- shapes 2-22, 7-10 to 7-11, 7-13
 - drawing 7-10, 7-13
 - filling 7-7, 7-8
 - filling with bitmap property 7-8
 - outlining 7-5
- shared property groups 51-12
- Shared Property Manager 51-12
- sharing forms and dialogs 2-35 to 2-37
- Shift states 7-23
- short strings 3-24
- ShortCut property 5-19
- shortcuts
 - adding to menus 5-19
- ShortString 3-24
- Show method 5-6, 5-7
- ShowAccelChar property 2-20
- ShowButtons property 2-18
- ShowFocus property 26-29
- ShowHint property 2-21
- ShowHints property 26-31
- ShowLines property 2-18
- ShowModal method 5-5
- ShowRoot property 2-18
- signalling events 8-9
- simple types 33-2
 - allowed in CORBA interfaces 28-6
- single document interface 4-1 to 4-2
- single-tiered applications 12-2, 12-6, 12-8, 13-1 to 13-17
 - BDE-based vs. flat-file 13-14
 - flat-file 13-13 to 13-17
 - vs. two-tiered applications 13-3
- Size property 19-13
- skeletons 28-2, 28-2 to 28-3, 28-7
 - marshaling 28-2, 28-7
- slow processes
 - using threads 8-1
- Smart Agents 28-2, 28-3
 - configuring 28-17 to 28-19
 - locating 28-3
 - starting 28-17
- socket components 30-5 to 30-8
- socket connections 30-2 to 30-3
 - closing 30-6, 30-8
 - endpoints 30-3, 30-5
 - multiple 30-5
 - opening 30-6, 30-7
 - sending/receiving information 30-10
 - types 30-2
- socket dispatcher
 - application 14-12, 14-20
- socket streams 30-12
- SocketHandle property 30-6, 30-7
- sockets 4-10, 30-1 to 30-14

- accepting client requests 30-3
- assigning hosts 30-4
- describing 30-3
- error handling 30-8
- event handling 30-8 to 30-10, 30-11, 30-12
- implementing services 30-1 to 30-2, 30-6, 30-7
- network addresses 30-3, 30-4
- providing information 30-4
- reading from 30-11
- reading/writing 30-10 to 30-14
- writing to 30-11
- software license requirements 11-10
- sort order 10-9
 - indexes 13-15, 24-6, 24-7
 - setting 20-9, 20-10
- Sorted property 2-17, 26-12
- sorting data 20-9 to 20-11
 - with alternative indexes 20-9
- source code
 - viewing
 - specific event handlers 2-25
- source datasets, defined 20-20
- source files
 - packages 9-2, 9-7, 9-8, 9-13
- Spacing property 2-15
- SparseCols property 27-9
- SparseRows property 27-9
- speed buttons 2-15
 - adding to toolbars 5-29 to 5-30
 - assigning glyphs 5-29
 - centering 5-29
 - engaging as toggles 5-30
 - event handlers 7-12
 - for drawing tools 7-12
 - grouping 5-30
 - initial state, setting 5-30
 - operational modes 5-29
- splitters 2-14
- SPX/IPX protocol 17-7
- SQL 12-2
 - ADO 23-20
- SQL applications
 - accessing tables 20-2
 - appending records 18-22
 - batch moves 20-23
 - data constraints 19-21, 19-22
 - deleting records 20-16
 - editing data 18-7
 - inserting records 18-7, 18-22
 - locating data 20-6, 20-8
 - sorting data 20-10
- SQL Builder 21-7
- SQL Explorer 14-2, 22-16
 - defining attribute sets 19-14
- SQL Links 11-4, 13-5
 - deploying 11-5, 11-11
 - driver files 11-5
 - drivers 17-7
 - installing 13-8
- SQL parser 21-16
- SQL property 21-5, 21-7, 21-8
 - changing 21-14
- SQL queries 21-1
 - creating 21-4, 21-6
 - at runtime 21-7
 - defining statements 21-5 to 21-8
 - multi-table 21-14
 - optimizing 21-12, 21-15
 - overview 21-1 to 21-5
 - parameter substitution 25-14, 25-18
 - preparing 21-13, 21-14
 - remote servers 21-17
 - result sets 21-13, 21-16 to 21-17
 - cursors and 21-15
 - getting at runtime 21-12
 - updating 21-17, 25-21
 - running 21-12 to 21-13, 25-19
 - from text files 21-8
 - setting parameters 21-8 to 21-11
 - at runtime 21-10
 - special characters and 21-6
 - submitting statements 21-13
 - update objects and 25-11, 25-13
 - whitespace characters and 21-6
- SQL servers 12-2, 16-11
 - constraints 19-21
- SQL standards 15-10, 21-4
 - remote servers 21-17
- SQL statements
 - ADO 13-13, 23-26
 - client supplied 24-17
 - client-supplied 15-4
 - decision datasets and 27-5
 - passthrough SQL 13-8
 - SQLPASSTHRUMODE 13-8
 - squares
 - drawing 40-9
- standard components 2-10
- standard events 34-4, 34-4 to 34-6
 - customizing 34-6
- Standard page (Component palette) 2-10
- StartTransaction method 13-6
- state information
 - communicating 14-26 to 14-27, 15-5
 - mouse events 7-23
- State property 2-16, 23-5, 23-14
 - grid columns 26-17
 - grids 26-16, 26-19
- stateless objects 51-8
- static binding 14-24, 28-11
 - COM 44-13
- static methods 32-7
- static text 2-20
- static text component 2-20
- status bars 2-20
 - internationalizing 10-8
- status constants
 - cached updates 25-10
- status information 2-20
- StatusCode property 29-15
- STDVCL40.DLL 11-6
- Step property 2-21
- StepBy method 2-21
- StepIt method 2-21
- stopping threads 8-10
- storage media 2-33
- stored directive 33-10
- stored procedures 12-5, 12-13
 - adding 22-3
 - ADO-based 12-14
 - creating 22-4
 - Interbase 12-14
 - overloaded 22-17
 - parameters 22-10
 - preparing 22-5
 - running 22-5
- StoreDefs property 20-17
- StoredProc Parameters editor 22-4
 - activating 22-16
 - setting parameters 22-13
 - viewing parameters 22-13
- StoredProcName property 22-3
- StrByteType 3-27
- streams 2-33
- Stretch property 26-11
- StretchDraw method 7-4, 36-3, 36-7
- string fields

- entering data 19-15
- size 19-7
- string grids 2-22
- String List editor 21-6
- string lists 2-28 to 2-32
 - adding objects 6-12 to 6-13
 - adding to 2-31
 - associated objects 2-32
 - copying 2-32
 - creating 2-28 to 2-30
 - deleting strings 2-31
 - finding strings 2-31
 - iterating through 2-31
 - loading from files 2-28
 - long-term 2-29
 - moving strings 2-31
 - owner-draw controls 6-12 to 6-13
 - position in 2-30, 2-31
 - saving to files 2-28
 - short-term 2-29
 - sorting 2-31
 - substrings 2-31
- string operators 3-32
- string reserved word 3-24
 - default type 3-23
 - VCL property types 3-25
- strings 3-23, 33-2, 33-8
 - 2-byte conversions 10-2
 - associating graphics 6-13
 - comparing 18-19
 - compiler directives 3-31
 - declaring and
 - initializing 3-28
 - extended character sets 3-32
 - files 3-39
 - local variables 3-30, 3-31
 - long 3-24
 - memory corruption 3-32
 - mixing and converting
 - types 3-30
 - PChar conversions 3-30
 - reference counting
 - issues 3-24, 3-30
 - returning 33-8
 - routines
 - case sensitivity 3-27
 - Multi-byte character support 3-27
 - runtime library 3-26
 - Windows locale 3-27
 - size 6-8
 - sorting 10-9
 - starting position 6-8
 - translating 10-2, 10-7, 10-9
 - truncating 10-3
 - types overview 3-23
 - variable parameters 3-32

Strings property 2-30

Structured Query Language 12-2

stThreadBlocking constant 30-10

stub-and-skeleton unit 28-6, 28-7, 28-11

stubs 28-2 to 28-3, 28-7, 28-12

 - creating 28-12
 - marshaling 28-2

Style property 2-17, 2-22, 14-37

 - brushes 7-8
 - combo boxes 26-12
 - owner-draw variants 6-12
 - pens 7-5
 - tool buttons 5-32

StyleRule property 14-37

Styles property 14-37

StylesFile property 14-37

stylesheets 14-36

subclassing Windows

 - controls 31-4
 - submenus 5-19

Subtotals property 27-12

summary values 27-19

 - cross-tabs 27-2, 27-3
 - decision graphs 27-15
 - maintained aggregates 24-11

support services 1-3

Suspend method 8-10

suspending threads 8-10

Sybase driver

 - deploying 11-5

Synchronize method 8-4

synchronizing data 20-24

 - on multiple forms 26-6, 26-7

System page (Component palette) 2-10

system registry 2-32

system resources,

 - conserving 31-4

T

tab controls 2-19

tab order 2-12

tab sets 2-19

table components 12-13, 20-2

Table HTML tag (<TABLE>) 29-18

table producers 29-23 to 29-24

 - setting properties 29-23

TableAttributes property 29-23

TableName property 20-3, 23-20

tables 12-13, 20-1

 - access rights 20-4
 - adding 20-2 to 20-4
 - ADO-based 12-14
 - closing 20-4
 - creating 13-10, 13-14, 20-17
 - decision support components and 27-3
 - deleting 20-16
 - displaying in grids 26-17
 - emptying 20-16
 - field and index
 - definitions 20-17
 - inserting records 18-21 to 18-22, 18-24
 - Interbase 12-14
 - master/detail
 - relationships 20-24 to 20-26
 - naming 20-3
 - non-database grids 2-21
 - opening 20-4
 - read-only 20-5
 - removing records 20-16
 - caution 20-16
 - renaming 20-16
 - restructuring 13-10
 - retrieving data 20-11 to 20-16, 20-18, 21-1
 - searching 20-5 to 20-8
 - sorting data 20-9 to 20-11
 - with alternative indexes 20-9
 - specifying type 20-3
 - synchronizing 20-24
 - updating data with 25-20

TableType property 20-3

TabOrder property 2-12

tabs

 - draw-item events 6-15
 - owner-draw styles 6-12

Tabs property 2-19

TabStop property 2-12

tabular display (grids) 2-21

tabular grids 26-27

TAction 5-35

TActionLink 5-35

TActionList 5-35

TActiveXControl 48-1

TADOCCommand 23-1, 23-22, 23-26, 23-27

TADOCConnection 23-1, 23-3, 23-5, 23-11, 23-13, 23-18, 23-19, 23-21, 23-22, 23-23

- connecting to a data store 23-3
- TADODataset 13-12, 23-18
- TADOQuery 13-12, 23-20, 23-26
- TADOStoredProc 13-12, 23-22
- TADOTable 13-12, 23-19, 23-20
- TADTField 19-1
- Tag property 19-13
- TAny 28-13
 - creating structured types 28-14
- TApplicationEvents 5-3
- TArrayField 19-1
- TAutoIncField 19-1
- TBatchMove
 - error handling 20-23
- TBCDField 19-1
- TBDEDataSet 18-26, 18-29
- TBitmap 36-4
- TBlobField 19-1
- TBlobStream 2-33
- TBooleanField 19-1
- TBrush 2-22
- fbCheck constant 5-32
- TBytesField 19-1, 19-2
- TCalendar 41-1
- TCGIApplication 29-5
- TCGIRequest 29-5
- TCGIResponse 29-5
- TCharProperty type 38-7
- TClassProperty type 38-7
- TClientDataSet 24-1
 - flat-file applications 13-13
- TClientSocket 30-5
- TClientWinSocket 30-6
- TColorProperty type 38-7
- TComObject
 - aggregation 3-20
- TComponent 2-6, 2-9, 2-10, 31-5
- TComponentProperty type 38-7
- TControl 2-11, 31-4, 34-5
- TCoolBand 2-16
- TCoolBar 5-28
- TCorbaConnection 14-22
- TCorbaDataModule 14-5
- TCorbaPrincipal 28-16
- TCP/IP 30-1
 - application servers and 14-12
 - clients 30-5
 - connecting to application server 14-20
 - multi-tiered applications 14-9
 - protocol 17-7
 - servers 30-6
- TCurrencyField 19-1
- TCustomContentProducer 29-17
- TCustomControl 31-4
- TCustomGrid 41-1, 41-2
- TCustomIniFile 2-33
- TCustomListBox 31-3
- TDatabase 17-1, 17-9
 - DatabaseName property and 13-3
 - temporary instances 16-7, 16-8, 17-2
- TDataSet 5-41, 18-2, 18-27
- TDataSetAction 5-41
- TDataSetCancel 5-41
- TDataSetDelete 5-41
- TDataSetEdit 5-41
- TDataSetFirst 5-41
- TDataSetInsert 5-41
- TDataSetLast 5-41
- TDataSetNext 5-41
- TDataSetPost 5-41
- TDataSetPrior 5-41
- TDataSetProvider 14-16, 15-1
- TDataSetTableProducer 29-24
- TDataSource 26-5 to 26-8
 - properties 26-6 to 26-7
- TDateField 19-1, 19-15
- TDateTime type 41-5
- TDateTimeField 19-2, 19-15
- TDBChart 12-12
- TDBCheckBox 26-2, 26-14
- TDBComboBox 26-2, 26-12
- TDBCtrlGrid 26-2, 26-27 to 26-29
 - properties 26-28
- TDBDataSet 18-26
 - properties 18-28
- TDBEdit 26-2, 26-9
- TDBGrid 26-2, 26-16
 - events 26-27
 - properties 26-21, 26-24
- TDBGridColumn 26-16
- TDBImage 26-2, 26-10
- TDBListBox 26-2, 26-11
- TDBLookupComboBox 26-2, 26-12 to 26-14
- TDBLookupListBox 26-2, 26-12 to 26-14
- TDBMemo 26-2, 26-9
- TDBNavigator 18-9, 18-10, 26-2, 26-29 to 26-31
- TDBRadioGroup 26-2, 26-15
- TDBRichEdit 26-10
- TDBText 26-2, 26-8
- TDCOMConnection 14-19
- TDecisionCube 27-4, 27-7 to 27-8
 - events 27-7
- TDecisionDrawState 27-12
- TDecisionGraph 27-2, 27-13
 - instantiating 27-13
- TDecisionGrid 27-2, 27-10
 - events 27-12
 - instantiating 27-10
 - properties 27-12
- TDecisionPivot 27-2, 27-3, 27-9
 - properties 27-10
- TDecisionQuery 27-4, 27-6
 - properties 27-7
- TDecisionSource 27-9
 - events 27-9
 - properties 27-9
- TDefaultEditor 38-12
- TDependency_object 4-7
- TDragObject 6-3
- technical support 1-3
- TEditAction 5-39
- TEditCopy 5-39
- TEditCut 5-39
- TEditPaste 5-39
- templates 2-35, 2-36
 - component 5-11, 5-12
 - decision graphs 27-16
 - menus 5-17, 5-23, 5-24 to 5-26
 - loading 5-24
 - Web applications 29-7
- templates, programming 4-2
- temporary files 16-13
- temporary objects 36-6
- TEnumProperty type 38-7
- Terminate method 8-5
- Terminated property 8-5
- terminating connections 17-8
- testing
 - components 31-12, 43-6
 - values 33-6
- TEvent 8-9
- text 26-9, 26-10
 - copying, cutting, pasting 6-9
 - deleting 6-9
 - drawing on canvases 7-23
 - in controls 6-6
 - internationalizing 10-8
 - owner-draw controls 6-11
 - printing 2-13
 - reading right to left 10-5
 - searching for 2-13

- selecting 6-8, 6-8 to 6-9
- truncated 26-9
- working with 6-6 to 6-11
- text controls 2-12 to 2-13
- text files
 - running queries from 21-8
- Text property 2-12, 2-13, 2-17, 2-20
- TextHeight method 7-4, 36-3
- TextOut method 7-4, 7-23, 36-3
- TextRect method 7-4, 36-3
- TextWidth method 7-4, 36-3
- TField 19-1
 - adding 19-1 to 19-5
 - events 19-16
 - methods 19-17
 - properties 19-3, 19-12 to 19-15
 - runtime 19-13
- TFieldDataLink 42-4
- TFile 3-37
- TFileStream 2-33
 - file I/O 3-37 to 3-40
- TFloatField 19-2
- TFloatProperty type 38-7
- TFontNameProperty type 38-7
- TFontProperty type 38-7
- TForm
 - scroll-bar properties 2-14
- TForm component 2-3
- TFrame 5-12
- TGraphic 36-4
- TGraphicControl 31-4, 40-2
- thin client applications 14-2, 14-28
- thread function 8-3
- thread objects 8-1
 - defining 8-2
 - initializing 8-2
 - limitations 8-2
- Thread Status box 8-13
- thread-aware objects 8-4
- ThreadID property 8-13
- threading models 8-12
 - ActiveForms 48-7
 - ActiveX controls 48-4
 - COM objects 45-2, 45-3 to 45-6
 - CORBA data modules 14-16
 - CORBA objects 28-5
 - MTS 51-15, 51-16
 - MTS data modules 14-14
 - remote data modules 14-13
- thread-local variables 8-5
- OnTerminate event 8-6
- threads 8-1 to 8-13
 - avoiding simultaneous access 8-6
 - blocking execution 8-6
 - caching 30-14
 - client sockets 30-11, 30-12 to 30-13
 - COM 8-11
 - coordinating 8-4, 8-6 to 8-10
 - CORBA 8-12
 - creating 8-10
 - critical sections 8-7
 - data access components 8-4
 - database sessions and 13-4, 16-2
 - distributed objects 8-11
 - executing 8-10
 - freeing 8-2, 8-3
 - graphics objects 8-5
 - ids 8-13
 - initializing 8-2
 - in-process servers 8-12
 - ISAPI/NSAPI programs 29-7, 29-21
 - libraries 8-12
 - limits on number 8-10
 - locking objects 8-6
 - message loop and 8-4, 8-11
 - message-based servers 8-11
 - priorities 8-1, 8-2
 - overriding 8-10
 - process space 8-3
 - returning values 8-8
 - server sockets 30-11, 30-13 to 30-14
 - service 4-5
 - stopping 8-10
 - terminating 8-5
 - using lists 8-5
 - using with client sockets 30-13
 - using with server sockets 30-14
 - VCL thread 8-4
 - waiting for 8-8
 - multiple 8-9
 - waiting for events 8-9
 - thread-safe objects 8-4
 - threadvar 8-5
- three-tiered applications *See* multi-tiered applications
- THTMLTableAttributes 29-23
- THTMLTableColumn 29-24
- TickMarks property 2-14
- TickStyle property 2-14
- TIcon 36-4
- tiDirtyRead constant 13-7
- tiers 14-1
- TImage
 - in frames 5-14
- TImageList 5-31
- time
 - internationalizing 10-9
- time fields 19-15
 - formatting values 19-16
- timeout events 8-10
- timer events 26-5
- times
 - entering 2-18
- TIniFile 2-33
- TIntegerField 19-2
- TIntegerProperty type 38-7, 38-9
- TInterfacedObject 3-20
 - deriving from 3-17
 - dynamic binding 3-17
 - implementing IUnknown 3-17
- tiReadCommitted constant 13-7
- tiRepeatableRead constant 13-7
- TISAPIApplication 29-5
- TISAPIRequest 29-5
- TISAPIResponse 29-5
- Title property
 - data grids 26-22
- TKeyPressEvent type 34-3
- TLabel 31-4
- .TLB files 44-13, 50-36
- TLIBIMP 44-14
- TListBox 31-3
- TMemIniFile 2-33
- TMemoField 19-2
- TMemoryStream 2-33
- TMessage 37-4, 37-6
- TMetafile 36-4
- TMethodProperty type 38-7
- TMsg 5-4
- TMTSDataModule 14-5
- TMultiReadExclusiveWriteSynchronizer 8-7
- TNotifyEvent 34-7
- TNumericField 19-2
- TObject 2-6, 32-3
- toggles 5-30, 5-32
- tool buttons 5-31
 - adding images 5-31
 - disabling 5-32
 - engaging as toggles 5-32
 - getting help with 5-34
 - grouping/ungrouping 5-32

- in multiple rows 5-32
- initial state, setting 5-32
- wrapping 5-32
- toolbars 2-16, 5-28
 - adding 5-31 to 5-32
 - adding panels as 5-29 to 5-30
 - context menus 5-34
 - default drawing tool 5-30
 - designing 5-28 to 5-35
 - disabling buttons 5-32
 - hiding 5-34
 - inserting buttons 5-29 to 5-30, 5-31
 - setting margins 5-30
 - speed buttons 2-15
 - transparent 5-32, 5-33
- tool-tip help 2-21
- Top property 2-11, 5-3, 5-29
- TopRow property 2-21
- TOrdinalProperty type 38-7
- TPageProducer 29-17
- TPanel 5-28
- TParameter 23-17, 23-28
- TPersistFormat type 23-16
- tpHigher constant 8-3
- tpHighest constant 8-3
- TPicture type 36-4
- tpIdle constant 8-3
- tpLower constant 8-3
- tpLowest constant 8-3
- tpNormal constant 8-3
- TPopupMenu 5-34
- TPropertyAttributes 38-10
- TPropertyEditor class 38-7
- tpTimeCritical constant 8-3
- TQuery 2-34, 21-1
 - adding 21-4
 - decision datasets and 27-5
- TQueryTableProducer 29-24
- track bars 2-14
- transaction attributes
 - MTS data modules 14-15
 - MTS objects 51-15, 51-17
- transactions 12-3 to 12-4, 13-5 to 13-9
 - cached updates and 25-1, 25-4
 - client-side support 51-19
 - committing 13-7
 - controlling 13-6 to 13-9
 - duration 13-6
 - implicit 13-5
 - isolation levels 13-7 to 13-8
 - local 13-9
 - MTS 14-6, 14-25, 51-5, 51-7 to 51-11
 - attributes 51-7
 - client-controlled 51-10
 - timeouts 51-10
 - MTS data modules 14-15
 - multi-tiered
 - applications 14-25
 - rolling back 13-7
 - server-side support 51-20
 - starting 13-6
 - uncommitted changes and 13-7
 - using databases 13-6
 - transfer records 43-2
 - Translolation property 13-7
 - translating character strings 10-2, 10-7, 10-9
 - 2-byte conversions 10-2
 - translation 10-8
 - Transliterate property 19-13, 20-20
 - transparent backgrounds 10-8
 - Transparent property 2-20
 - transparent toolbars 5-32, 5-33
 - TReader 3-37
 - tree views 2-18
 - TReferenceField 19-2
 - TRegistry 2-32
 - TRegistryIniFile 2-33
 - TRegSvr 11-3, 44-14
 - TRemoteDataModule 14-5
 - triangles 7-11
 - triggers 12-5
 - truncated text 26-9
 - try reserved word 36-6, 43-5
 - TScrollBar 2-14
 - TSearchRec 3-33
 - TServerClientThread 30-11
 - TServerClientWinSocket 30-7
 - TServerSocket 30-6
 - TServerWinSocket 30-7
 - TService_object 4-7
 - TSession 16-1, 17-9
 - adding 16-3, 16-16
 - TSessionList 16-1
 - TSessions component 16-1
 - TSetElementProperty type 38-7
 - TSetProperty type 38-7
 - TSmallintField 19-2
 - TSocketConnection 14-20
 - TStoredProc 22-3
 - TStream 2-33
 - TStringField 19-2, 19-15
 - TStringList 2-28 to 2-32
 - TStringProperty type 38-7
 - TStrings 2-28 to 2-32
 - TStringStream 2-33
 - TTable 2-34, 20-1
 - decision datasets and 27-5
 - TThread 8-2
 - TThreadList 8-5, 8-6
 - TTimeField 19-2, 19-15
 - TToolBar 5-28, 5-31
 - TToolButton 5-28
 - TTypedComObject
 - type library requirement 44-13
 - TUpdateAction type 25-25
 - TUpdateKind type 25-24
 - TUpdateSQL 21-17, 25-11
 - events 25-22 to 25-23
 - TVarBytesField 19-2
 - TWebActionItem 29-7
 - TWebApplication 29-5
 - TWebRequest 29-5
 - TWebResponse 29-5, 29-7
 - TWinCGIRequest 29-5
 - TWinCGIResponse 29-5
 - TWinControl 10-8, 31-3, 34-5
 - TWindowAction 5-40
 - TWindowArrange 5-40
 - TWindowCascade 5-40
 - TWindowClose 5-40
 - TWindowMinimizeAll 5-40
 - TWindowTileHorizontal 5-40
 - TWindowTileVertical 5-40
 - TWinSocketStream 2-33, 30-12
 - two-byte character codes 10-2
 - two-phase commit 14-25
 - TWordField 19-2
 - two-tiered applications 12-3, 12-6, 12-8, 13-1 to 13-10
 - vs. one-tiered applications 13-3
 - TWriter 3-37
 - type declarations
 - enumerated types 7-11
 - objects and 2-7
 - properties 40-3
 - type information 44-12
 - type interfaces 44-13
 - type libraries 44-9, 44-11, 44-12 to 44-14, 50-1 to 50-36
 - accessing 44-13
 - ActiveX controls 48-2
 - adding interfaces 50-32
 - adding methods 50-32
 - adding properties 50-32
 - attributes 50-8

- benefits 44-14
 - browsers 44-14
 - browsing 44-14
 - contents 44-12, 50-1
 - creating 44-12, 50-23 to 50-36
 - creating Automation
 - controllers 46-1
 - deploying 50-36
 - exporting as IDL 50-35
 - IDL and ODL 44-12
 - including as resources 45-3, 50-36
 - opening existing 50-31
 - referencing other type libraries 50-8
 - registering 44-14, 50-35
 - registering objects 44-14
 - saving 50-35
 - supplying help 50-6
 - syntax checking 50-35
 - tools 44-14
 - type information 50-7 to 50-23
 - uninstalling 44-14
 - unregistering 44-14
 - valid types 50-23
 - when to use 44-13
 - type libraries, optimize performance 50-11
 - Type Library editor 50-2 to 50-35
 - adding CoClasses 50-33
 - adding enumerated types 50-33
 - adding interfaces 50-32
 - adding methods 50-32
 - adding properties 50-32
 - aliases 50-19, 50-29
 - application servers 14-16
 - attributes page 50-6
 - CoClasses 50-16 to 50-17, 50-28
 - CORBA interfaces 28-5
 - creating type libraries 50-23 to 50-36
 - dispatch interfaces 50-14 to 50-15, 50-28
 - enumerated types 50-18, 50-29
 - errors 50-5
 - flags page 50-7
 - help attributes 50-7
 - interfaces 50-9 to 50-14, 50-27
 - main elements 50-3
 - modules 50-21 to 50-23, 50-30
 - Object list pane 50-5
 - Object Pascal vs. IDL 50-23, 50-25 to 50-31
 - opening libraries 50-31
 - records 50-19 to 50-20, 50-29
 - status bar 50-5
 - syntax 50-7, 50-23, 50-25 to 50-31
 - syntax checking 50-35
 - text page 50-7
 - toolbar 50-3
 - type information 50-7 to 50-23
 - type information pages 50-6
 - unions 50-20 to 50-21, 50-30
 - Uses page 50-8
 - Type Library, Dispinterfaces 50-14
 - type reserved word 7-12
 - typeinfo 50-2
 - types
 - Automation 47-9
 - Char 10-2
 - CORBA objects 28-6
 - mapping to database tables 20-22
 - message-record 37-6
 - properties 33-2, 33-8, 38-8
 - type libraries 50-23
 - user-defined 40-3
- ## U
-
- Unassociate Attributes
 - command 19-15
 - uncommitted changes 13-7
 - unDeleted constant 25-10
 - undeleting cached records 25-8 to 25-10
 - undoing changes 18-23
 - Unicode characters 10-3
 - strings 3-25, 3-26
 - Unicode standard
 - strings 3-23
 - UniDirectional property 21-15
 - unindexed tables 18-22, 18-24
 - searching 20-6
 - unions
 - Type Library editor 50-20 to 50-21, 50-30
 - units
 - accessing from other units 2-7
 - adding components 31-10
 - existing
 - adding a component 31-10
 - including packages 9-3
 - property editors 38-7
 - Unlock method 8-6
 - UnlockList method 8-6
 - UnPrepare method 21-14
 - unpreparing queries 21-14
 - UnregisterPooled
 - procedure 14-7
 - UnRegisterTypeLib function
 - uninstalling type libraries 44-14
 - update errors
 - resolving 15-6, 15-9, 24-21, 24-21 to 24-22
 - response messages 14-35
 - Update method
 - actions 5-39
 - update objects 25-11
 - applying 25-18
 - event handling 25-22 to 25-23
 - executing statements 25-19
 - preparing SQL statements 25-13
 - Update SQL editor 25-13
 - UPDATE statements 21-12, 21-13, 25-11
 - UpdateCalendar method 42-3
 - UpdateMode property 15-8
 - UpdateObject property 18-29, 25-11
 - typecasting 25-17
 - UpdateRecordTypes
 - property 18-29, 25-8, 25-9
 - UpdatesPending
 - property 18-29, 25-3
 - UpdateStatus property 18-29, 25-10
 - UpdateTarget method 5-39
 - updating data 14-34 to 14-35
 - affecting multiple records 15-4
 - delta packets 15-6
 - multi-tiered
 - applications 15-3, 15-5, 15-7 to 15-9
 - identifying tables 15-9
 - screening updates 15-9
 - reconciling update errors 24-21 to 24-22
 - updating records 14-34 to 14-35, 15-6, 26-7

- batch operations 20-21
- multiple datasets 25-5
- multi-tiered clients 24-20 to 24-22
- queries and 21-17
- reconciling updates 24-21
- up-down controls 2-14
- URLs
 - URLs vs. 29-2
- URL property 29-13
- URLs 29-2
 - host names 30-4
 - IP addresses 30-4
 - javascript libraries 14-31, 14-32
 - URIs vs. 29-2
 - Web browsers 29-3
 - Web connections 14-21
- Use (Object Repository) 2-36
- Use Unit command (File) 2-34, 5-2
- USEPACKAGE macro 9-8
- user commands 5-35, 5-36
- user interfaces 12-11 to 12-15
 - forms 5-1 to 5-2
 - isolating 12-7
 - layout 5-3 to 5-4
 - multi-record 12-12
 - single record 12-11
- user-defined messages 37-5, 37-7
- user-defined types 40-3
- uses clause 2-7
 - adding data modules 2-34
 - avoiding circular references 5-2
 - including packages 9-3
- usInserted constant 25-10
- usModified constant 25-10
- usUnmodified constant 25-10

V

- \$V compiler directive 3-32
- validating data entry 19-17
- value expressions 19-21
- Value property 19-18
- ValueChecked property 26-15
- values 33-2
 - Boolean 33-2, 33-9, 42-3
 - default data 19-21, 26-11
 - default property 33-7, 33-9 to 33-10
 - redefining 39-2, 39-3
 - null 18-24, 20-13
 - testing 33-6

- Values property 26-15
- ValueUnchecked property 26-15
- var reserved word
 - event handlers 34-3
- variables 21-6
 - declaring
 - example 2-7
 - object 2-7
 - objects and 2-7 to 2-8
 - variants 18-15, 19-20
 - CORBA 28-13
 - TAny 28-13
- VCL 31-1 to 31-2
 - main thread 8-4
 - overview 2-1
 - string properties 3-25
- VCL40 package 9-1, 9-9
 - PENWIN.DLL 9-12
- version information
 - ActiveX controls 48-5, 48-7
- vertical track bars 2-14
- VertScrollBar 2-14
- video cassettes 7-31
- video clips 7-28, 7-30
- virtual
 - directive 32-8
 - method tables 32-8
 - methods 32-8, 35-4
 - properties as 33-2
 - property editors 38-8 to 38-9
- visibility 2-7
- Visible property 19-13
 - cool bars 5-34
 - menus 5-26
 - toolbars 5-34
- VisibleButtons property 26-30, 26-31
- VisibleColCount property 2-21
- VisibleRowCount property 2-21
- VisiBroker ORB 14-12
- Visual cross-process
 - objects 44-9
- VisualSpeller Control 11-3
- vtable binding
 - early binding
 - COM 44-13
- vtables 44-4
 - COM interface pointer 44-4

W

- WaitFor method 8-8, 8-9
- WaitForData method 30-12
- WantReturns property 2-13

- WantTabs property 2-13
 - memo fields 26-9
 - rich edit controls 26-10
- .WAV files 7-31
- \$WEAKPACKAGEUNIT
 - compiler directive 9-11
- Web applications
 - ActiveX 44-11
 - database 14-28 to 14-38
 - deploying 11-7
 - object 29-7
- Web browsers 29-3
 - URLs 29-3
- Web connections 14-9, 14-21
- Web deployment 48-17 to 48-25
 - multi-tiered
 - applications 14-29
 - options 48-20
- Web Deployment Options dialog box 48-18
- Web dispatcher 29-6, 29-8 to 29-10
 - auto-dispatching
 - objects 14-34, 29-9
 - DLL-based applications and 29-7
 - handling requests 29-7, 29-12
 - selecting action items 29-10, 29-11
- Web items 14-35
 - properties 14-36 to 14-37
- Web modules 29-6 to 29-7, 29-8
 - adding database sessions 29-21
 - DLLs and, caution 29-7
- Web page editor 14-35 to 14-36
- Web pages 29-3
 - MIDAS page producer 14-35 to 14-38
- Web server applications
 - using MTS for
 - debugging 29-26 to 29-27
- Web server applications 4-10, 29-1 to 29-29
 - accessing databases 29-21
 - adding to projects 29-7
 - converting 29-28
 - creating 29-6
 - creating responses 29-12
 - debugging 29-25 to 29-29
 - event handling 29-9, 29-11, 29-12
 - managing database connections 29-21

- MIDAS 14-30 to 14-38
 - overview 29-5 to 29-8
 - posting data to 29-14
 - querying tables 29-24
 - resource locations 29-2
 - response templates 29-18
 - sending files 29-17
 - standards 29-1
 - templates 29-7
 - types 29-5
 - Web dispatcher and 29-8
 - Web servers 14-29
 - client requests and 29-4
 - Web site (Delphi support) 1-3
 - WebDispatch property 14-34
 - WebPageItems property 14-35
 - whitespace characters
 - running queries on 21-6
 - wide character strings 47-9
 - wide characters 10-3
 - runtime library routines 3-26
 - WideChar 3-23, 3-25
 - WideString 3-25
 - Width property 2-11, 2-20, 5-3, 26-17, 26-22
 - pens 7-5, 7-6
 - TScreen 11-8
 - Win 3.1 page (Component palette) 2-10
 - Win32 page (Component palette) 2-10
 - Win-CGI programs 4-11, 29-4, 29-5
 - creating 29-6
 - debugging 29-28
 - INI files 29-5
 - window
 - class 31-4
 - controls 31-3
 - handles 31-3, 31-5
 - message handling 41-4
 - procedures 37-2, 37-3
 - Windows
 - API functions 31-3, 36-1
 - common dialog boxes 43-1
 - creating 43-2
 - executing 43-4
 - controls, subclassing 31-4
 - device contexts 31-7, 36-1
 - events 34-4
 - Graphics Device Interface (GDI) 7-1
 - messages 5-4, 37-2
 - pen width support 7-6
 - windows
 - resizing 2-14
 - Windows NT
 - debugging Web server applications 29-25
 - Windows socket objects 30-5
 - client sockets 30-6
 - clients 30-6
 - server sockets 30-7
 - wizards 2-35, 8-12
 - COM object 45-1, 45-2
 - Component 31-8
 - CORBA Data Module 14-15 to 14-16, 28-4
 - CORBA Object 28-4
 - MTS Data Module 14-14 to 14-15
 - MTS Object 51-15
 - Remote Data Module 14-13 to 14-14
 - Resource DLL 10-9
 - WM_APP constant 37-5
 - WM_KEYDOWN message 42-8
 - WM_LBUTTONDOWN
 - message 42-8
 - WM_MBUTTONDOWN
 - message 42-8
 - WM_PAINT messages 7-2
 - WM_RBUTTONDOWN
 - message 42-8
 - WM_SIZE message 41-4
 - WndProc method 37-3, 37-4
 - word wrapping 6-7
 - WordWrap property 2-13, 6-6, 39-1
 - memo fields 26-9
 - wordwrapping 6-7
 - wParam parameter 37-2
 - wrAbandoned constant 8-9
 - Wrap property 5-32
 - Wrapable property 5-32
 - wrappers 31-4, 43-2
 - initializing 43-3
 - wrError constant 8-9
 - Write By Reference, type library property 50-11
 - Write method
 - TFileStream 3-38
 - write method 33-6
 - write reserved word 33-8, 40-4
 - WriteBuffer method
 - TFileStream 3-38
 - write-only properties 33-6
 - writing property settings 33-6, 38-8
 - wrSignaled constant 8-9
 - wrTimeout constant 8-9
- ## X
-
- \$X compiler directive 3-32
 - Xerox Network System (XNS) 30-1
 - XML brokers 14-33 to 14-35
 - XML files 23-16
 - XMLBroker property 14-36
 - XMLDataSetField property 14-36
- ## Y
-
- Year property 41-5
- ## Z
-
- Z compiler directive 9-12

