

# **Borland® Delphi 8 for .NET™**

---

Borland Software Corporation 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© 2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3259

# Concepts

---

## Getting Started

### What's Delphi for .NET

Defining Requirements .....	17
Modeling Applications .....	17
Designing User Interfaces .....	17
Generating and Editing Code .....	17
Compiling, Debugging, and Deploying Applications .....	17
Controlling Access and Tracking Changes to Code .....	18
The .NET Framework .....	18

### Tour of the IDE

Welcome Page .....	19
Forms .....	19
Designer Surface .....	21
Tool Palette .....	21
Object Inspector .....	22
Object Repository .....	22
Project Manager .....	23
Data Explorer .....	24
Code Editor .....	24
To-Do Lists .....	25

### Starting a Project

Type of Projects .....	26
Additional Projects .....	29
Unmanaged Code and COM/Interop .....	29

## Managing the Development Cycle

### Managing the Development Cycle Overview

Source Control Integration .....	31
User Interface Design .....	31
Code Visualization .....	32
Build, Compile, Run, and Debug .....	32

### Using Source Control

How Delphi 8 for .NET Interacts with Source Control Systems .....	33
Source Control Basics .....	34
Configuring the Source Control System .....	35
Repository Basics .....	35
Working with Projects .....	36
Working with Files .....	36
Synchronizing Files .....	36

### Designing User Interfaces

Using the Designer .....	37
Setting Designer Options .....	37

### Using Code Visualization

Code Visualization and UML Static Structure Diagrams .....	38
Understanding the Relationship between Source Code and Code Visualization .....	38

Compiling and Building Applications	
Compiler Options .....	40
Compiler Status and Information .....	40
Compiler Errors .....	41
Using Translation Tools	
Satellite Assembly Wizard .....	42
Translation Manager .....	42
Translation Repository .....	42
Files Generated by the Translation Tools .....	43
Debugging Applications	
Stepping Through Code .....	44
Evaluate/Modify .....	44
Breakpoints .....	44
Watches .....	45
Debug Windows .....	45
Remote Debugging .....	45
Deploying Applications	
Simple Applications .....	46
Applications that Include Shared Assemblies .....	46
Installation Programs .....	46
Redistributing Delphi 8 for .NET Files .....	46
Redistributing the .NET Framework .....	47
Redistributing Third Party Software .....	47
<b>Modeling with Delphi for .NET</b>	
Modeling Tools Overview	
Model-Powered Applications and the ECO Framework .....	50
Basic UML Concepts .....	50
Introduction to the ECO Framework .....	52
Integrated Modeling Tools in Delphi 8 for .NET .....	58
<b>Working with Unmanaged Code</b>	
Using COM Interop in Managed Applications	
COM Interop Overview .....	62
COM Interop Terminology .....	63
COM Interop Tools in the .NET Framework SDK .....	65
Using COM Interop Assemblies in the IDE .....	66
Deploying Applications That Use COM Interop .....	67
Using Platform Invoke with Delphi for .NET	
Calling Unmanaged Functions .....	69
Structures .....	72
Callback Functions .....	75
Passing Object References .....	75
Using COM Interfaces .....	77
<b>Building Web Applications with ASP.NET</b>	
ASP.NET Overview	
ASP.NET Architecture .....	81
Web Forms Components .....	82
Web Forms Data Access .....	82

Web Services .....	82
ASP.NET Namespace .....	83
ASP.NET Application Deployment .....	83
DB Web Controls for ASP.NET	
DB Web Controls Architecture .....	84
Data-aware Components Advantages .....	84
Supported Data Access Components .....	85
DB Web ControlsNamespace .....	85
ASP.NET Application Deployment with DB Web Controls .....	85
<b>Building Web Services with ASP.NET</b>	
ASP.NET Web Services Overview	
Web Service Architecture .....	87
Web Service Prerequisites .....	88
Web Service Scenarios .....	88
ASP.NET Web Services Files .....	89
Web Services Protocol Stack	
Web Services Protocol Stack .....	91
Transport Layer .....	92
XML Messaging .....	92
WSDL Layer .....	92
Service Discovery .....	93
ASP.NET Web Services Support	
ASP.NET Web Services Client Support .....	94
ASP.NET Web Services Server Support .....	95
ASP.NET Web Services Namespaces .....	95
<b>Building Windows Applications with Windows Forms</b>	
Windows Forms Overview	
Windows Forms Architecture .....	97
Windows Forms .....	97
Windows Forms Components .....	98
Windows Forms Data Access .....	98
Windows Forms Namespace .....	98
Windows Forms Application Deployment .....	98
<b>Building Database Applications with ADO.NET</b>	
ADO.NET Overview	
ADO.NET Architecture .....	100
ADO.NET User Interfaces .....	102
BDP.NET Namespace .....	102
ADO.NET Application Deployment .....	103
Borland Data Providers for Microsoft .NET	
Data Provider Architecture .....	104
BDP.NET Advantages .....	105
BDP.NET and ADO.NET Components .....	105
BDP.NET Data Types .....	105
BDP.NET Interfaces .....	105
BDP.NET Data Types	
BDP.NET and .NET Framework .....	106

Data Types .....	106
DB2 .....	106
Interbase .....	107
MS SQL and MSDE .....	107
Oracle .....	108
<b>BDP.NET Component Designers</b>	
Component Designer Relationships .....	111
Connection Editor .....	111
Command Text Editor .....	111
Generate DataSets .....	112
Configure Data Adapter .....	112
Data Explorer .....	112
<b>Delphi for .NET Database Technologies</b>	
Building .NET Applications with TADONETConnector .....	113
Building .NET Applications with dbExpress.NET .....	113
Building .NET Applications with IBX.NET .....	114
Building .NET Applications with the DataSnap .NET Client (DCOM) .....	114
Building .NET Applications with BDE.NET .....	114
<b>Getting Started with IBX Components</b>	
IBX components .....	115
<b>Building VCL Applications with VCL for .NET</b>	
<b>VCL for .NET Overview</b>	
What is VCL? .....	124
The Relationship between VCL.NET and the .NET Framework .....	125
VCL.NET Components .....	125
Borland.VCL Namespace .....	127
Porting Delphi Applications to Delphi 8 for .NET .....	127
Importing .NET Components for Use in VCL.NET Applications .....	128
<b>Porting Applications to VCL for .NET</b>	
General Language Issues .....	129
New Language Features .....	129
Porting Web Service Client Applications .....	130
<b>Building Reports in Delphi for Microsoft .NET</b>	
<b>Rave Reports Overview</b>	
Creating New Reports in Delphi 8 for .NET .....	132
Using Rave Reports ActiveX Components .....	132

# Procedures

---

## Getting Started

Adding Components to a Form .....	135
Adding References .....	136
Adding and Removing Files .....	137
Adding Templates to the Object Repository .....	138
Copying References to a Local Path .....	139
Creating a Component Template .....	140
Creating a Project .....	142
Customizing the Form .....	143
Customizing Toolbars .....	144
Customizing the Tool Palette .....	145
Docking Tool Windows .....	146
Filtering Searched Components .....	147
Exploring .NET Assembly Metadata .....	148
Exploring Windows Type Libraries .....	149
Installing Custom Components .....	150
Renaming Files using the Project Manager .....	151
Saving Desktop Layouts .....	152
Setting Component Properties .....	153
Setting Dynamic Properties .....	154
Setting Project Options .....	156
Setting Properties and Events .....	157
Setting Tool Preferences .....	158
Using To-Do Lists .....	159
Writing Event Handlers .....	161

## ASP.NET

Building an ASP.NET Database Application .....	163
Building an ASP.NET "Hello world" Application .....	167
Building an ASP.NET Application .....	169
Building an Application with .....	171
Extending .....	173
Using the HTML Tag Editor .....	178

## Compiling and Building Applications

Adding Languages to a Project .....	182
Building Packages .....	184
Editing Resource Files in the Translation Manager .....	186
Setting Up the External Translation Manager .....	188

## Database

Adding a New Connection to the Data Explorer .....	191
Browsing a Database in the Data Explorer .....	192
Building a Windows Forms Database Application .....	193
Creating Database Projects from the Data Explorer .....	196
Creating Table Mappings .....	197

Executing SQL in the Data Explorer .....	200
Handling Errors in Table Mapping .....	201
Modifying Connections in the Data Explorer .....	203
Passing Parameters in a Database Application .....	206
Using the Data Adapter Preview .....	210
Using the CommandText Designer .....	211
Using the Data Adapter Designer .....	212
Using the Connection Editor Designer .....	213
Using Standard DataSets .....	215
Using Typed DataSets .....	220
<b>Debugging Applications</b>	
Adding a Watch .....	224
Attaching to a Running Process .....	225
Setting and Modifying Breakpoints .....	226
Inspecting and Changing the Value of Data Elements .....	230
Resolving internal errors .....	232
Modifying Variable Expressions .....	235
Preparing a Project for Debugging .....	236
Debugging Remote Applications .....	237
<b>Editing Code</b>	
Using Code Folding .....	241
Customizing Code Editor .....	242
Recording a Keystroke Macro .....	243
Using Code Insight .....	244
Using Code Snippets .....	246
<b>Modeling</b>	
Adding Columns to a Component .....	248
Building an ECO-Enabled User Interface .....	249
Building Applications with the ECO Framework .....	252
Deploying an ECO-Enabled Application .....	253
Exporting a Code Visualization Diagram to an Image .....	254
Importing and Exporting a Model Using XML .....	255
Using the Model View Window and Code Visualization Diagram .....	256
Using the Model View Window and ECO Class Diagram .....	258
Using the ECO Space Designer .....	261
Using the ECO Wizards .....	265
Using the OCL Expression Editor .....	268
Using the Overview Window .....	270
<b>Source Control</b>	
Adding Files to the Source Control Project .....	272
Checking In Files .....	273
Checking Out Files .....	274
Using the Commit Browser .....	275
Configuring Source Control Providers .....	278
Connecting to the Source Control Repository .....	279
Placing a Project into Source Control .....	280

Pulling a Project from Source Control .....	282
Removing Files from Source Control .....	285
Undoing a Check Out Operation .....	286
Running an SCC Application .....	287

## **VCL for .NET**

Building VCL Forms Applications With Graphics .....	289
Building a VCL Forms MDI Application Without Using a Wizard .....	290
Building a VCL Forms MDI Application Using a Wizard .....	295
Building a VCL Forms SDI Application .....	296
Building a Network Application with Socket Components .....	297
Building a VCL Forms dbExpress.NET Database Application .....	299
Building a VCL Forms ADO.NET Database Application .....	302
Creating Actions in a VCL Forms Application .....	305
Building a VCL Forms "Hello world" Application .....	308
Using ActionManager to Create Actions in a VCL Forms Application .....	310
Building a VCL Forms Application .....	312
Building an Application with XML Components .....	313
Displaying a Bitmap Image in a VCL Forms Application .....	317
Drawing a Polygon in a VCL Forms Application .....	319
Drawing Rectangles and Ellipses in a VCL Forms Application .....	320
Drawing a Rounded Rectangle in a VCL Forms Application .....	321
Drawing Straight Lines In a VCL Forms Application .....	322
Building a Multithreaded Application .....	324
Writing Cleanup Code .....	325
Avoiding Simultaneous Thread Access to the Same Memory .....	326
Defining the Thread Object .....	328
Handling Exceptions .....	331
Using the Main VCL Thread .....	332
Waiting for Threads .....	334
Writing the Thread Function .....	336
Placing A Bitmap Image in a Control in a VCL Forms Application .....	337
Creating a VCL Forms ActiveX Active Form .....	339
Creating a VCL Forms ActiveX Button .....	342
Importing .NET Controls to VCL.NET .....	345



# Reference

---

## Delphi Overview

Delphi Overview	
Program Organization .....	349
Example Programs .....	351

## Programs and Units

Programs and Units	
Program Structure and Syntax .....	356
Unit Structure and Syntax .....	358
Unit References and the Uses Clause .....	360
Using Namespaces (.NET Only)	
Declaring Namespaces .....	365
Searching Namespaces .....	366
Using Namespaces .....	368

## Syntactic Elements

Syntactic Elements	
The Delphi Character Set .....	371
The Delphi Character Set and Basic Syntax .....	371
Comments and Compiler Directives .....	376
Declarations and Statements	
Declarations .....	377
Statements .....	379
Simple Statements .....	379
Structured Statements .....	381
Blocks and Scope .....	390
Expressions	
Expressions .....	393
Function Calls .....	401
Set Constructors .....	402
Indexes .....	402
Typecasts .....	402

## Data Types, Variables, and Constants

Data Types, Variables, and Constants	
About Types .....	407
Simple Types	
Ordinal Types .....	409
Real Types .....	416
String Types	
About String Types .....	419
Short Strings .....	420
Long Strings .....	421
WideString .....	421
Working with null-Terminated Strings .....	422
Structured Types	
Sets .....	426

Arrays .....	427
Records .....	432
File Types .....	435
Pointer Types	
Overview of pointers .....	437
Pointer Types .....	438
Procedural Types	
About Procedural Types .....	441
Procedural Types in Statements and Expressions .....	442
Variant Types	
Variants Overview .....	445
Variant Type Conversions .....	445
Variants in Expressions .....	447
Variant Arrays .....	447
OleVariant .....	448
Type Compatibility	
Type Identity .....	449
Type Compatibility .....	450
Assignment Compatibility .....	450
Declaring Types	
Type Declaration Syntax .....	452
Variables	
Declaring Variables .....	454
Declared Constants	
True Constants .....	457
Typed Constants .....	459
<b>Procedures and Functions</b>	
Procedures and Functions	
About Procedures and Functions .....	464
Declaring Procedures and Functions .....	464
Calling Conventions .....	467
Forward and Interface Declarations .....	468
External Declarations .....	469
Overloading Procedures and Functions .....	471
Local Declarations .....	473
Parameters	
About Parameters .....	475
Parameter Semantics .....	475
String Parameters .....	479
Array Parameters .....	480
Default Parameters .....	483
Calling Procedures and Functions	
Program Control and Parameters .....	485
Open Array Constructors .....	485
<b>Classes and Objects</b>	
Classes and Objects	
Class Types .....	488

Inheritance and Scope .....	490
Visibility of Class Members .....	491
Forward Declarations and Mutually Dependent Classes .....	494
Fields	
About Fields .....	496
Class Fields (.NET) .....	497
Methods	
About Methods .....	498
Method Binding .....	500
Overloading Methods .....	504
Constructors .....	505
The Class Constructor (.NET) .....	506
Destructors .....	507
Message Methods .....	508
Properties	
About Properties .....	510
Property Access .....	510
Array Properties .....	512
Index Specifiers .....	514
Storage Specifiers .....	515
Property Overrides and Redclarations .....	515
Class Properties (.NET) .....	517
Class References	
Class-Reference Types .....	519
Class Operators .....	520
Class Methods .....	521
Exceptions	
About Exceptions .....	523
When To Use Exceptions .....	523
Declaring Exception Types .....	524
Raising and Handling Exceptions .....	525
Standard Exception Classes and Routines .....	530
<b>Standard Routines and I/O</b>	
Standard Routines and I/O	
File Input and Output .....	532
Text File Device Drivers .....	535
Handling null-Terminated Strings .....	537
Other Standard Routines .....	538
<b>Libraries and Packages</b>	
Libraries and Packages	
Calling Dynamically Loadable Libraries .....	543
Static Loading .....	543
Dynamic Loading .....	543
Writing Dynamically Loadable Libraries	
Using Export Clause in Libraries .....	545
Library Initialization Code .....	547
Global Variables in a Library .....	548

Libraries and System Variables .....	548
Exceptions and Runtime Errors in Libraries .....	549
Shared-Memory Manager (Win32 Only) .....	549
<b>Packages</b>	
Understanding Packages .....	550
Package Declarations and Source Files .....	550
Compiling Packages .....	552
<b>Object Interfaces</b>	
<b>Object Interfaces</b>	
Interface Types .....	556
Interface and Inheritance .....	557
Interface Identification .....	557
Calling Conventions for Interfaces .....	558
Interface Properties .....	559
Forward Declarations .....	559
<b>Implementing Interfaces</b>	
Class Declarations .....	560
Method Resolution Clause .....	561
Changing Inherited Implementations .....	562
Implementing Interfaces by Delegation .....	562
<b>Interface References</b>	
Implementing Interface References .....	565
Interface Assignment Compatibility .....	566
Interface Typecasts .....	566
Interface Querying .....	567
<b>Automation Objects</b>	
Dispatch Interface Types .....	568
Dispatch interface methods .....	568
Dispatch interface properties .....	569
Accessing Automation Objects .....	569
Automation Object Method-Call Syntax .....	569
Dual Interfaces .....	570
<b>Memory Management</b>	
<b>Memory Management</b>	
The Memory Manager (Win32 Only) .....	572
Variables .....	572
<b>Internal Data Formats</b>	
Integer Types .....	574
Character Types .....	574
Boolean Types .....	574
Enumerated Types .....	574
Real Types .....	574
Pointer Types .....	576
Short String Types .....	577
Long String Types .....	577
Wide String Types .....	577
Set Types .....	578

Static Array Types .....	578
Dynamic Array Types .....	578
Record Types .....	579
File Types .....	580
Procedural Types .....	582
Class Types .....	582
Class Reference Types .....	583
Variant Types .....	583
Memory Management Issues on the .NET Platform	
Constructors .....	505
Finalization .....	584
The Dispose Pattern .....	585
Unit Initialization and Finalization .....	586
Unit Initialization Considerations for Assemblies and Dynamically Linked Packages .....	587
<b>Program Control</b>	
Program Control	
Passing Parameters .....	590
Handling Function Results .....	592
Handling Method Calls .....	592
Understanding Exit Procedures .....	593
<b>Inline Assembly Code (Win32 Only)</b>	
Inline Assembly Code	
Using the asm Statement .....	596
Using Registers .....	596
Assembler Syntax	
Assembler Statement Syntax .....	597
Labels .....	597
Instruction Opcodes .....	598
Assembly Directives .....	599
Operands .....	603
Expressions	
Differences between Delphi and Assembler Expressions .....	605
Expression Elements .....	606
Expression Classes .....	610
Expression Types .....	612
Expression Operators .....	613
Assembly Procedures and Functions	
Compiler Optimizations .....	616
Function Results .....	617
<b>.NET Topics</b>	
Using .NET Custom Attributes	
Declaring a Custom Attribute Class .....	619
Using Custom Attributes .....	620
Using the DllImport Custom Attribute .....	621
Custom Attributes and Interfaces .....	621
<b>Command-Line Switches</b>	

IDE command-line switches and options	
IDE command-line switches .....	623
General options .....	623
Debugger options .....	624
Project options .....	624

# Concepts

# Getting Started

---

The Delphi 8 for .NET integrated development environment (IDE) provides many tools and features to help you build powerful .NET applications quickly. Not all features and tools are available in all editions of Delphi 8 for .NET. For a list of features and tools included in your edition, refer to the feature matrix on [www.borland.com/delphi\\_net](http://www.borland.com/delphi_net).



# What's Delphi 8 for .NET?

---

Delphi 8 for .NET is an integrated development environment (IDE) for building Delphi applications that run in the Microsoft .NET environment. The Delphi 8 for .NET IDE provides a comprehensive set of tools that streamline and simplify the development life cycle. The tools available in the IDE depend on the edition of Delphi 8 for .NET you are using. The following sections briefly describe these tools.

## Defining Requirements

---

Delphi 8 for .NET provides an interface to CaliberRM, a Web-based requirements definition and management system designed to help control the product development process. Within the IDE, you can access CaliberRM to collaborate on project requirements and ensure that your applications meets end-user needs.

## Modeling Applications

---

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

## Designing User Interfaces

---

The Delphi 8 for .NET visual designer surface lets you create graphical user interfaces by dragging and dropping components from the *Tool Palette* to a form. Using the designers, you can create Windows Forms, Web Forms, Janeva, and HTML pages.

## Generating and Editing Code

---

Delphi 8 for .NET auto-generates much of your application code as soon as you begin a project. To help you quickly complete the remaining application logic, the text-based *Code Editor* provides features such as code completion, reusable code snippets, recorded keystroke macros, and custom key mappings. Syntax highlighting and code folding make your code easier to read and navigate.

## Compiling, Debugging, and Deploying Applications

---

Within the IDE, you can set compiler options, compile and run your application, and view compiler messages. The integrated debugger lets you find and fix runtime and logic errors, control program execution, and step through code to watch variables and modify data values. The .NET Framework includes several utilities to help you prepare applications for deployment. Delphi 8 for .NET includes InstallShield Express for creating Windows Installer setups.

## Controlling Access and Tracking Changes to Code

---

Source control systems enable team development by controlling access and tracking changes to source code and other files. Delphi 8 for .NET uses the Microsoft Common Source Code Control API (SCC API) to provide a common interface to StarTeam, CVS, ClearCase, and Visual SourceSafe. Within the Delphi 8 for .NET IDE, you can perform common source control tasks, such as file check in, check out, and synchronization.

## The .NET Framework

---

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

Delphi 8 for .NET makes the entire Framework class library available in the IDE to help you develop .NET applications. Delphi 8 for .NET enhances the Framework in the following areas:

- The Delphi 8 for .NET Borland Data Providers for .NET provide access to InterBase, Oracle, DB2 Universal, and Microsoft SQL Server databases.
- Several database utilities assist in performing tasks such as connecting to databases, browsing and editing databases, and executing SQL queries.
- The .NET Menu Designers simplify the creation of main menus and context menus on Windows Forms.

# Tour of the IDE

---

When you start Delphi 8 for .NET, the integrated development environment (IDE) launches and displays several tools and menus. The IDE helps you visually design user interfaces, set object properties, write code, and view and manage your application in various ways.

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

The tools available in the IDE depend on the edition of Delphi 8 for .NET you are using and include the:

- Welcome Page
- Forms
- Designer Surface
- Tool Palette
- Object Inspector
- Object Repository
- Project Manager
- Data Explorer
- Code Editor
- To-Do Lists

The following sections describe each of these tools.

## Welcome Page

---

When you open Delphi 8 for .NET, the *Welcome Page* appears with a number of options and links to select from. As you develop a project or multiple projects, a table displays your most recent projects and a timestamp of when each project was modified. This provides quick access to your most recent projects.

If you close the *Welcome Page*, you can reopen it by choosing View ► Welcome Page.

The *Welcome Page* contains buttons to access and open projects. You can also access the latest online Help information by clicking the *Help* button. The *Project* button opens an existing project or project group. The *New* button opens the *Object Repository* and lets you select from a variety of project templates to create your desired .NET application. You can also choose File ► New ► Other and select the template most appropriate for your user interface design.

## Forms

---

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

Delphi 8 for .NET provides three types of forms, as described in the following sections. Select the form that best suits your application design, whether it's a Web application that provides business logic functionality over the Web, or a Windows application that provides processing and high-performance content display. To switch between the designer and *Code Editor*, click their associated tabs below the IDE.

To access forms, choose File ► New ► Other.

## Windows Forms

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

To access a Windows Form, choose File ► New ► Other and select the *Application* icon from the Projects directory.

## ASP.NET Web Forms

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

To access an ASP.NET Web Form, choose File ► New ► Other and select the *ASP.NET Web application* icon from the ASP Projects directory.

## Janeva

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

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

To access a Janeva, choose File ► New ► Other and select the *VCL.NET application* icon from the *Delphi for .NET Projects* directory.

## HTML Designer

Use the *HTML designer* to view and edit ASP.NET Web Forms or HTML pages. This designer provides a *Tag Editor* for editing HTML tags alongside the visual representation of the form or page. You can also use the *Object Inspector* to edit properties of the visible items on the HTML page and to display

the properties of any current HTML tag in the *Tag Editor*. A combo box located above the *Tag Editor* lets you display and edit SCRIPT tags.

To create a new HTML file, choose File ► New ► Other and select HTML page from the Markup File directory. When the HTML page is displayed, an Insert menu is added to the main menu. Use this menu to dynamically insert tables, images, user controls, and more.

## Designer Surface

---

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

## Visual Components

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

## Nonvisual Components and the Component Tray

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

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

## Tool Palette

---

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

## Customized Components

In addition to the components that are installed with Delphi 8 for .NET, you can add customized or third party components to the *Tool Palette* and save them in their own category.

## Component Templates

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

## Object Inspector

---

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

Use the *Properties* tab to change physical attributes of your components. Depending on your selection, some category options let you enter values in a text box while others require you to select values from a drop-down box. For Boolean operations, you toggle between True or False. After you change your components' physical attributes, you create event handlers that control how the components function.

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

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

## Object Repository

---

To simplify development, Delphi 8 for .NET offers predesigned templates, forms, and other items that you can easily access and use in your application. The *Object Repository* is accessible by choosing File ► New ► Other. A *New Items* dialog box appears. You can also edit or remove existing objects from the *Object Repository* by right-clicking the *Object Repository* to view your editing options.

**Note:** Although the dialog box is called *New Items*, this is actually the *Object Repository*.

## Inside the Object Repository

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

## Object Repository Templates

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

## Project Manager

---

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

## Add References

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

## Copy References to a Local Path

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

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

## Add Web References

You can quickly add a Web Reference to your client application and access the Web Service you want to use. When you add a Web Reference, you are importing a WSDL document into your client application, which describes a particular Web Service. Once you imported the WSDL document, Delphi

8 for .NET generates all the interfaces and class definitions you need for calling that Web Service. To use the Add Web Reference feature, from your *Project Manager*, right-click the *Web Services* node.

## Data Explorer

---

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

## Code Editor

---

As you design the user interface for your application, Delphi 8 for .NET generates the underlying code. When you select and modify the properties of your objects, your changes are automatically reflected in the source files. The *Code Editor* gives you the flexibility of writing additional code directly into the *Code Editor* because it's a full-feature ANSI editor.

Because all your programs share common characteristics, Borland Delphi 8 for .NET supplies autogenerated code to get you started. Do not modify the autogenerated code for the *Initialize Components* method. Doing so will cause your form to disappear when you click the *Design* tab. You can think of the autogenerated code as an outline that you can examine to create your program. Additionally, Delphi 8 for .NET provides supplemental tools to help you write code such as code insight, code browsing, code snippets, and code folding.

## Code Insight

*Code Insight* refers to a subset of features embedded in the *Code Editor* that aid in the code writing process. These features display context-sensitive pop-up windows and provide the following services:

- Help identify common statements you wish to insert into your code.
- Assist in the selection of properties and methods.
- Display events available for a particular class.
- Provide view declaration information for identifiers.

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

## Code Browsing

The *Code Editor* has browse buttons similar to those found in Web browsers. To display the browse buttons, right-click an empty area of the IDE toolbar and select *Browser*. As you move through your files, the *Code Editor* keeps track of where you have been in the code. You can click the drop-down arrow next to the browse buttons to move forward and backward through a history of these references.



## Code Snippets

Code snippets are commonly used programming statements, such as `if`, `while`, and `for` statements, that you can insert into your code. When the *Code Editor* is open, you can double-click a code snippet on the *Tool Palette* to add it to your code. You can also create your own code snippets by selecting code in the *Code Editor*, pressing the ALT key, and dragging the code to the *Tool Palette*.

## Code Folding

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

## To-Do Lists

---

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

# Starting a Project

---

A project is a collection of files which includes, but is not limited to, project files (*name.bdsproj*), assemblies (*System.dll*), program database files (*name.pdb*), optional resource files (*.html*, *.jpeg*, *.gif*), executables (*name.exe*), and many others that make up an application. Projects are either created at design time or generated when you compile the project source code. To assist in the development process, the *Object Repository* offers many predesigned templates, forms, files, and other items that you can use to create .NET applications.

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

This section includes information about:

- types of projects
- working with unmanaged code

## Type of Projects

---

Depending on the edition of Delphi 8 for .NET that you are using, you can create traditional Windows applications, ASP.NET Web applications, ADO.NET database applications, Web Services applications, and many others. Delphi 8 for .NET also supports assemblies, custom components, multi-threading, and COM. For a list of the features and tools included in your edition, refer to the feature matrix on [www.borland.com/delphi\\_net](http://www.borland.com/delphi_net).

## Windows Applications

You can create Windows applications using Windows Forms to provide processing and high-performance content display. Windows applications can function as a front end to ADO.NET databases.

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

## ASP.NET Web Applications

You can create Web applications using ASP.NET Web Forms to provide Web access to databases and Web Services. Web Forms provide the user interface for Web applications and consist of HTML, server controls, and application logic in code-behind files. Delphi 8 for .NET lets you drag and drop components and provides in-place HTML editing.

## ASP.NET Web Services Applications

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

Delphi 8 for .NET simplifies the creation of Web Services by providing methods for creating a SOAP Server application. The .asmx and .dlls files are created automatically and you can test the Web Service within the IDE, without writing a client application for it.

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

## VCL.NET Applications

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

Delphi 8 for .NET simplifies the task of building .NET-enabled applications by supporting VCL components that have been augmented to run on the .NET Framework. This eliminates the need for you to create custom components to provide standard VCL component capabilities. This makes the process of porting Win32 applications to .NET much simpler and reliable.

## Database Applications

Whether your application uses Windows Forms, Web Forms, or VCL Forms, Delphi 8 for .NET has several tools that make it easy to connect to a database, browse and edit a database, execute SQL queries, and display live data at design time.

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

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

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

## Model-Driven Applications

Modeling is a term used to describe the process of software design. Developing a model of a software system is roughly equivalent to an architect creating a set of blueprints for a large development project.

Like a set of blueprints, a model not only depicts the system as a whole, but also allows you to focus in on specifics such as structural and behavioral details. Abstracted away from any particular programming language (and at some levels, even from specific technology), the model allows all participants in the development cycle to communicate in the same language.

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

The ECO features include:

- Automatic mapping of the model classes, with their attributes and relationships, to a relational schema.
- Automatic evolution of schema when the model changes.
- Specification of the persistence backend. You can choose to store objects in a relational database or in an XML file.
- Design-time structural validation of the model and its Object Constraint Language (OCL) expressions.
- Runtime validation of the OCL expressions.
- An event mechanism that allows you to receive notifications whenever objects are added, changed, or removed.

Delphi 8 for .NET IDE leverages the ECO framework to provide an integrated surface on which to develop your application model. The IDE and its modeling surface features include:

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

**Note:** Not all modeling features are available in all editions of Delphi 8 for .NET. To determine the modeling features supported in your product edition, refer to the feature matrix on [www.borland.com/delphi\\_net](http://www.borland.com/delphi_net).

## Assemblies

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

## Additional Projects

---

In addition to the project types described above, Delphi 8 for .NET provides templates to create class libraries, control libraries, console applications, Visual Basic applications, reports, text files, and more. These templates are stored in the *Object Repository* and you can access them by choosing File ► New ► Other.

## Unmanaged Code and COM/Interop

---

Unmanaged code refers to applications that do not target the .NET Framework Common Language Runtime (CLR). COM/Interop is a .NET service that allows seamless interoperation between managed and unmanaged code. The COM/Interop service allows you to leverage existing COM servers and ActiveX controls in your .NET applications, and expose .NET components in legacy unmanaged applications. The Delphi 8 for .NET IDE includes tools to help you integrate your legacy COM servers and ActiveX controls into managed applications. Additionally, you can add references to unmanaged DLLs to your project, and then browse the types contained, just as you would with managed assemblies.

# Managing the Development Life Cycle

---

The application development life cycle is an iterative process that involves designing, developing, testing, debugging, and deploying applications. Delphi 8 for .NET provides powerful tools to support this iterative process, including integrated source control, form design tools, the Delphi for .NET compiler, an integrated debugging environment, and installation and deployment tools.

# Managing the Development Cycle Overview

---

The development cycle as described here is a subset of Application Lifecycle Management (ALM), dealing specifically with the part of the cycle that includes the implementation and control of actual development tasks. It does not include such things as modeling applications. Delphi 8 for .NET provides a framework of tools that helps you manage and perform all of your development requirements.

These tools include:

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

## Source Control Integration

---

Delphi 8 for .NET provides integration with a number of source control systems, including Borland StarTeam, Rational ClearCase, CVS, and Microsoft Visual SourceSafe. This integration allows you to access your source control system in one of two ways:

- Invoke the source control system in a separate process.
- Manage project files within the source control system from the Delphi 8 for .NET IDE.

Invoke the source control system in a separate process if you need to use specific features of that system, which are not exposed in the Delphi 8 for .NET IDE. The source control application appears in a separate window.

In most cases, you manage your project files from within the Delphi 8 for .NET IDE. The integration provided allows you to check-in, check-out, update, commit, and otherwise manage your source files using a simplified user interface. The integration supports the level of multi-user capabilities provided by your specific source control system.

## User Interface Design

---

Delphi 8 for .NET provides a rich environment for designing a .NET user interface. In addition to the Windows Form designer, which includes a full set of visual components, the IDE gives you tools to build ASP.NET Web Forms, along with a set of Web Controls. Delphi 8 for .NET also includes a VCL.NET Forms design tool, which allows you to build .NET applications using VCL components. The standard Designer offers a variety of alignment tools, font tools, and visual components for building many types of applications, including MDI and SDI applications, tabbed dialogs, and data aware applications.

## Code Visualization

---

The Code Visualization feature of Delphi 8 for .NET provides the means to document and debug your class designs using a visual paradigm. As you load your projects and code files, you can use the Model View window to get both a hierarchical graphical view of all of the objects represented in your classes, as well as a UML-like model of your application objects. This feature can help you visualize the relationships between objects in your application, and can assist you in developing and implementing.

## Build, Compile, Run, and Debug

---

Delphi 8 for .NET provides a full-featured build and compile system, along with an integrated debugger. The visual approach to building, compiling, and running your application makes the entire development process simpler than in the past. Projects with subprojects and multiple source files can be compiled all together, which is called *building*, or you can compile each project individually.

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

Because the entire set of processes is both menu and shortcut key driven, you can develop your own personal style of programming that is seamless and integrated as much as possible, making the entire development lifecycle effective and painless.



# Using Source Control

---

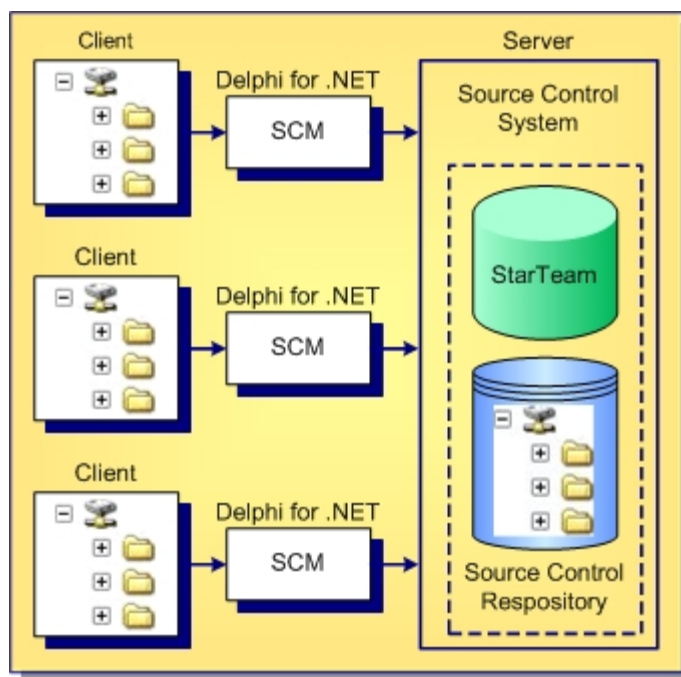
Borland's Delphi 8 for .NET provides integration with several source control systems, allowing you to perform the most common source control tasks from within the development environment. Additionally, you can write your own interfaces to unsupported source control systems, using the Microsoft SCC API.

The Delphi 8 for .NET development environment supports the following source code control systems:

- Borland StarTeam and the StarTeam Microsoft SCC Integration
- Concurrent Versions System (CVS) GUI implementations that support SCC API
- Rational ClearCase
- Microsoft Visual SourceSafe

## How Delphi 8 for .NET Interacts with Source Control Systems

---



Your source code control system typically consists of server and client components. On the server side, the system maintains a database repository that captures a complete snapshot of your project files and incremental changes (deltas or differences) to those files. On your local client, you use the source control client software to manage access to projects and files stored in the server repository, to maintain an audit trail of changes you make to the projects and files, and to provide you with conflict management capabilities. In Delphi 8 for .NET the Source Control Manager is an integrated feature that allows you

to connect to your source control system on a remote server, to place projects into and pull projects from your source control repository, and to check in, check out, merge, and compare files.

Delphi 8 for .NET uses the Microsoft SCC API to manage interaction with source control systems. You must use a system that includes an integration to the SCC API. Some products such as StarTeam, provide a separately installable SCC Integration application. Other products, like CVS, do not directly support the SCC API, but a variety of 3rd-party clients built as front-end applications to CVS do provide the integration. Any integration must support MSSCCI 1.1.

## Source Control Basics

---

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

Most source control systems adhere to a concept of a logical *project*, within which files are stored, usually in one or more tree directory structures. A source control system project might contain one or many Delphi 8 for .NET projects, in addition to other documents and artifacts. The system also enforces its own user authentication or, very often, takes advantage of the authentication provided by the underlying operating system. Doing so allows the source control system to maintain an audit trail or snapshot of updates to each file. These snapshots are typically referred to as *diffs*, for differences. By storing only the differences, the source control system can keep track of all changes with minimal storage requirements. When you want to see a complete copy of your file, the system performs a merge of the differences and presents you with a unified view. At the physical level, these differences are kept in separate files until you are ready to permanently merge your updates, at which time you can perform a *commit* action.

This approach allows you and other team members to work in parallel, simultaneously coding within multiple, shared projects, without the danger of code collision. Source control systems, in their most basic form, protect you from code conflicts and loss of early sources. Most source control systems give you the tools to manage code files with check in/check out capabilities, conflict reconciliation, and reporting capabilities. Most systems do not include logic conflict reconciliation or build management capabilities, although some products such as Rational ClearCase do provide build and workspace management capabilities. For details about your particular source control system capabilities, refer to the appropriate product documentation provided by your source control system vendor.

Commonly, source control systems only allow you to compare and merge updates to text files, including code files and other types of documents. The source control systems supported by Delphi 8 for .NET allow you to include binary files in the projects you place under control, but you cannot compare or merge multiple sources of those files. Typically, when you move binary files, or any files that are not textual, into your source control system, the system overwrites any existing copies of those files with newer sources. Resource files or other types of automatically generated binaries might also be overwritten or ignored by the source control system. If you need to maintain copies of different sources of these types of files, you might consider creating a manual tracking system for those files.

In Delphi 8 for .NET, you can access any supported and installed source control system from the Team menu.

## Configuring the Source Control System

---

Configuring Delphi 8 for .NET and your source control system to interact is a simple task. First, install and configure your source control system itself. If you are an individual developer or in a small shop, you might be required to perform this task yourself. Otherwise, if you are in a large, distributed environment, it is likely that your system administrator or project manager handles this task. In any case, refer to the documentation for your particular source control system to perform the installation and configuration before you attempt to use source control capabilities in Delphi 8 for .NET and to learn more about how your system operates. After your source control system is installed and configured, Delphi 8 for .NET auto-detects which system you have installed.

You can install and use multiple source control systems from within Delphi 8 for .NET. When you perform certain tasks you are required to log in to the source control system repository of your choice.

## Repository Basics

---

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

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

Some source control systems allow you to create a local repository in which you can maintain a snapshot of your projects. Over time the local image of your projects differs from the remote repository. You can establish a regular policy for merging and committing changes from your local repository to the remote repository. In a small shop, you can install the repository on your local machine and give shared access to that machine to other users. Generally, it is not safe to give each member of your team a separate repository on a shared project. If you are each working on completely separate projects and you want to keep each project under source control locally, you might use individual local repositories. Even in this case, however, you can create multiple repositories on a remote server, which provides centralized support, backup, and maintenance.

## Working with Projects

---

Source control systems, like development environments, use the project concept to organize and track groups of related files. No matter which source control system you use, you create a project that maintains your file definitions and locations. You also create projects in Delphi 8 for .NET to organize the various assemblies and source code files for any given application. Delphi 8 for .NET stores the project parameters in a project file. You can store the project file in your source control system project, in addition to the various code files you create. You might share your project file among all the developers on your team, or you might each maintain a separate project file. Most source control systems consider development environment project files to be binary, whether they are actually binary files or not. As a consequence, when you check a project file into a source control system repository in which the same project file or a project file with the same name is already stored, the source control system overwrites the older file with the newer one. The same is true when you pull a file, or check it out, from the repository onto your local system, if the same project file or a project file with the same name is located there.

## Working with Files

---

The file is the lowest-level object that you can manage in a source control system. Any code you want to maintain under source control must be contained in a file. Most source control systems store files in a logical tree structure. Some systems, such as CVS, actually use terms like branch, to refer to a directory level. You can create files in a Delphi 8 for .NET project and include them in your source control system, or you can pull existing files from the source control system. You can put an entire directory into the source control system, after which you can check out individual files, multiple files, or entire subdirectory trees. Delphi 8 for .NET gives you control over your files at two levels, both at the project level within Delphi 8 for .NET and in the source control system, through the Delphi 8 for .NET interface to the source control system.

## Synchronizing Files

---

One of the most powerful features of any source control system is its ability to synchronize multiple instances of the same file, to compare lines of text and mark those that overlap, then to merge the file instances without destroying conflicting lines of code or text. Most systems also provide the capability to update your local instance of a file or even an entire project's files with the latest source of those files stored in the source control system repository, without destroying any new lines of code you've added to the local instance of the file. You can perform these synchronization operations using the *Commit Browser* from within Delphi 8 for .NET.

# Designing User Interfaces

---

A graphical user interface (GUI) consists of one or more windows that let users interact with your application. At design time, those windows are called *forms*. Delphi 8 for .NET provides a designer for creating Windows Forms, Web Forms, Java, and HTML pages. The designer and forms help you create professional-looking user interfaces quickly and easily.

## Using the Designer

---

When you create a Windows, Web, or Web Services application, the IDE automatically displays the appropriate type of form on the *Design* tab in the IDE. As you drop components, such as labels and text boxes, from the *Tool Palette* on to the form, Delphi 8 for .NET generates the underlying code to support the application. You can use the *Object Inspector* to modify the properties of components and the form. The results of those changes appear automatically in the source code on the *Code* tab. Conversely, as you modify code with *Code Editor*, the changes you make are immediately reflected on the *Design* tab.

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

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

## Setting Designer Options

---

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

# Code Visualization Overview

---

The Code Visualization feature is available in both the Enterprise and Architect versions of Delphi 8 for .NET. All other modeling tools and information related to modeling relates only to the Architect version of Delphi 8 for .NET.

## Code Visualization and UML Static Structure Diagrams

---

Delphi 8 for .NET's code visualization diagram presents a graphical view of your source code, which is reflected directly from the code itself. When you make changes in source code, the graphical depiction on the diagram is updated automatically. The code visualization diagram corresponds to a UML *static structure* diagram. A structural view of your project focuses on UML packages, data types such as classes and interfaces, and their attributes, properties, and operations. A static structure diagram also shows the relationships that exist between these entities.

This section will explain the relationship between source code and the code visualization diagram.

**Note:** code visualization, and the integrated UML modeling tools are two separate and distinct features of Delphi 8 for .NET. Code visualization refers to the ability to scan an arbitrary set of source code, and map the declarations within that code onto UML notation. The resulting diagram is "live", in the sense that it always reflects the current state of the source code, but you cannot make changes directly to the code visualization diagram itself. Delphi 8 for .NET's model driven UML tools go a step further, giving you the ability to design your application on the diagramming surface. While both product features are based on Borland Together technologies, they each use different underlying mechanisms to produce and manipulate the diagram. The integrated model design surface is additionally based on the design-time capabilities of Borland's Enterprise Core Objects (ECO) framework. This document covers only the code visualization diagram; please use the link at the bottom of this topic for more information on the ECO framework and UML modeling tools.

## Understanding the Relationship between Source Code and Code Visualization

---

Delphi 8 for .NET's code visualization tools use the UML notation and conventions to graphically depict the elements declared in source code. The *Code Visualization diagram* shows you the logical relationships, or *static structure* in UML terms, of the classes, interfaces and other types defined in your project. The IDE creates the *Code Visualization diagram* by mapping certain source code constructs (such as class declarations, and implementation of interfaces) onto their UML counterparts, which are then displayed on the diagram.

## Top-Level Organization: Projects, UML Packages, and .NET Namespaces

To begin, code visualization consists of two parts of the IDE working together: The *Model View window*, and the *Code Visualization diagram*. The *Model View window* shows you the logical structure

of your projects in a tree, as opposed to the file-centric view of the *Project Manager window*. Each project in a project group is a top-level node in the *Model View tree*.

Nested within each project tree-node, you will find UML packages. Each UML package corresponds to a .NET namespace declaration in your source code (.NET namespaces can span multiple source files). You can expand the UML package to reveal the types declared within.

## Inheritance and Interface Implementation

The UML term for the relationship formed when one class inherits from a superclass is *generalization*. When the IDE sees an inheritance relationship in your source code, it creates a generalization link within the child class node in the *Model View tree*. On the *Code Visualization diagram*, the generalization link will be shown the using standard UML notation of a solid line with a hollow arrowhead pointing at the superclass.

The UML term for interface implementation is *realization*. Similar to the case of inheritance, the IDE creates a realization link when it sees a class declaration that implements an interface. The realization link appears within the implementor class in the *Model View tree*, and on the diagram as a dotted line with a hollow arrowhead pointing at the interface. There will be one such realization link for every interface implemented by the class.

## Associations

In the UML, an *association* is a navigational link produced when one class holds a reference to another class (for example, as an attribute or property). Code visualization creates association links when one class contains an attribute or property that is a non-primitive data type. On the diagram the association link exists between the class containing the non-primitive member, and the data type of that member.

## Class Members: Attributes, Operations, Properties, and Nested Types

Code visualization can also map class and interface member declarations to their UML equivalents. Within the elements on the *Code Visualization diagram*, members are grouped into four distinct categories:

- Fields: Contains field declarations. The type, and optional default value assignment are shown on the diagram.
- Methods: Contains method declarations. Visibility, scope, and return value are shown.
- Properties: Contains Delphi property declarations. The type of the property is shown.
- Classes: Contains nested class type declarations.

Standard UML syntax is used to display the UML declaration of attributes, operations, and properties. Each of the four categories can be independently expanded or collapsed to show or hide the members within.

# Compiling, Building, and Running Applications

---

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

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

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

## Compiler Options

---

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

On the *Compiler* page, you can also save compiler options as an *option set*. This lets you quickly change options based on your development activity. For example, you can set compiler options specific to debugging your project, and then change the option set when you are done debugging it.

If you need to specify additional compiler options, you can invoke the compiler from the command line. For a complete list of the compiler options and information about running the compiler from the command line, see the *Language Guide* section of this Help system.

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

## Compiler Status and Information

---

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

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



## Compiler Errors

---

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

# Using Translation Tools

---

Delphi 8 for .NET includes a suite of Translation Tools to facilitate localization and development of .NET applications for different locales. The Translation Tools include the following:

- Satellite Assembly Wizard
- Translation Manager
- Translation Repository

## Satellite Assembly Wizard

---

You can make your application adapt to the locale of the system it is running on, simply by providing the appropriate satellite assemblies. Satellite assemblies are used to deploy language-specific resources for an application. When your application starts up, it checks the locale of the local system. If it finds any satellite assemblies with the same name as the EXE, assembly, or package files it is using, it checks the extension on those assemblies. 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, assemblies, or package.

The Satellite Assembly Wizard lets you add languages to your project and creates a satellite assembly for each of those languages. You can add new satellite assemblies to a project at any time. If you have multiple projects open in the IDE, you can process several at once. The Satellite Assembly Wizard can also help you remove and restore languages.

## Translation Manager

---

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

The External Translation Manager (ETM) is a version of the Translation Manager that you can set up and use outside of the IDE. The ETM has the same functionality as the Translation Manager, with some additional menus and toolbars.

## Translation Repository

---

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

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

The Translation Repository stores data in XML format. By default, the file is named default.tmx and is located in the Delphi 8 for .NET\bin directory.

## Files Generated by the Translation Tools

---

The files generated by the Translation Tools include:

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

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

# Debugging Applications

---

The integrated debugger lets you find and fix both runtime errors and logic errors in your Delphi 8 for .NET application. Using the debugger, you can step through code, set breakpoints and watches, and inspect and modify program values. As you debug your application, the debug windows are available to help you manage the debug session and provide information about the state of your application. The remote debug server lets you debug a Delphi 8 for .NET application running on a remote computer.

## Stepping Through Code

---

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

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

## Evaluate/Modify

---

The Evaluate/Modify functionality allows you to evaluate an expression for which you've set a breakpoint. You can also pass expression values to the currently running process. For instance, you can modify a value for a variable and insert that value into the variable, which you might want to do if that value is to be passed to another method at some point during the execution of the application. This allows you to provide values in-process, which you might otherwise not be able to do. The Evaluate/Modify functionality is customized to whichever implementation language you are using and that is supported by the product.

## Breakpoints

---

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

## Watches

---

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

## Debug Windows

---

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

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

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

## Remote Debugging

---

Remote debugging lets you debug a Delphi 8 for .NET .NET application running on a remote computer. Your computer must be connected to the remote computer through TCP/IP. After you install and run the remote debug server, and copy the required application files to the remote computer, you can use Delphi 8 for .NET to connect to that computer and begin debugging.

# Deploying Applications

---

This section provides general information about deploying .NET applications. For more detailed information, see the .NET Framework SDK online Help, or the topics listed at the end of this section.

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

## Simple Applications

---

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

## Applications that Include Shared Assemblies

---

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

## Installation Programs

---

For more complex applications that consist of multiple files, you can use an installation tool, such as InstallShield Express. InstallShield Express can be installed from the Delphi 8 for .NET installation CD. After installing it, refer to the online InstallShield online Help for information about using the product.

## Redistributing Delphi 8 for .NET Files

---

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

However, if you build the application by compiling the application to have external references to VCL for .NET assemblies, the application will have external dependencies on the .NET Framework, the

Borland.Delphi.dll, and whatever Delphi 8 for .NET packages you have added to the project references, for example, Borland.VclRtl.dll or Borland.Vcl.dll.

Some of the files that are associated with Delphi 8 for .NET applications are subject to redistribution limitations or cannot be redistributed at all. Refer to the following documents for the legal stipulations regarding the redistribution of these files.

File	Description
Deploy.txt	Contains deployment considerations for each edition of Delphi 8 for .NET.
License.txt	Addresses legal rights and obligations concerning Delphi 8 for .NET.
Readme.txt	Contains last minute information about Delphi 8 for .NET, possibly including information that could affect the redistribution rights for Delphi 8 for .NET files.

These files are located, by default, at C:\Program Files\Borland\BDS\2.0.

## Redistributing the .NET Framework

If you plan to deploy your application to a computer that does not have the .NET Framework installed on it, you will need to redistribute and install the .NET Framework with your application. Microsoft provides a redistributable installer called dotnetfx.exe, which contains the common language runtime and .NET Framework components required to run .NET applications. For more information about dotnetfx.exe, see the .NET Framework SDK online Help.

## Redistributing Third Party Software

The redistribution rights for third party software, such as components, utilities, and helper applications, are governed by the vendor that supplies the software. Before you redistribute any third party software with your Delphi 8 for .NET application, consult the third party vendor or software documentation for information regarding redistribution.

# Modeling with Delphi 8 for .NET

---

Delphi 8 for .NET's integrated modeling tools tie together the processes of design and development. The class diagramming tools integrated into the IDE are based on well known industry standards such as UML and OCL. The Enterprise Core Object (ECO) framework implements the UML version 1.4 Metamodel, and leverages the .NET framework to make the model available at both designtime and runtime. This section provides an overview of UML concepts and terminology, the ECO framework, and the IDE's built in graphical modeling tools.

Please note that the Code Visualization feature is available in both Enterprise and Architect versions of the product. As such, any information about this feature refers to both versions. All other modeling features relate only to the Architect version of Delphi 8 for .NET and the information contained herein refers only to modeling tools and features in the Architect version.



# Modeling Tools Overview

---

With the exception of the Code Visualization feature, all material regarding modeling and modeling tools herein relates only to the Architect version of Delphi 8 for .NET. The information about Code Visualization refers to both the Enterprise and Architect versions of Delphi 8 for .NET. This topic contains material adapted from the *Together ControlCenter User Guide*.

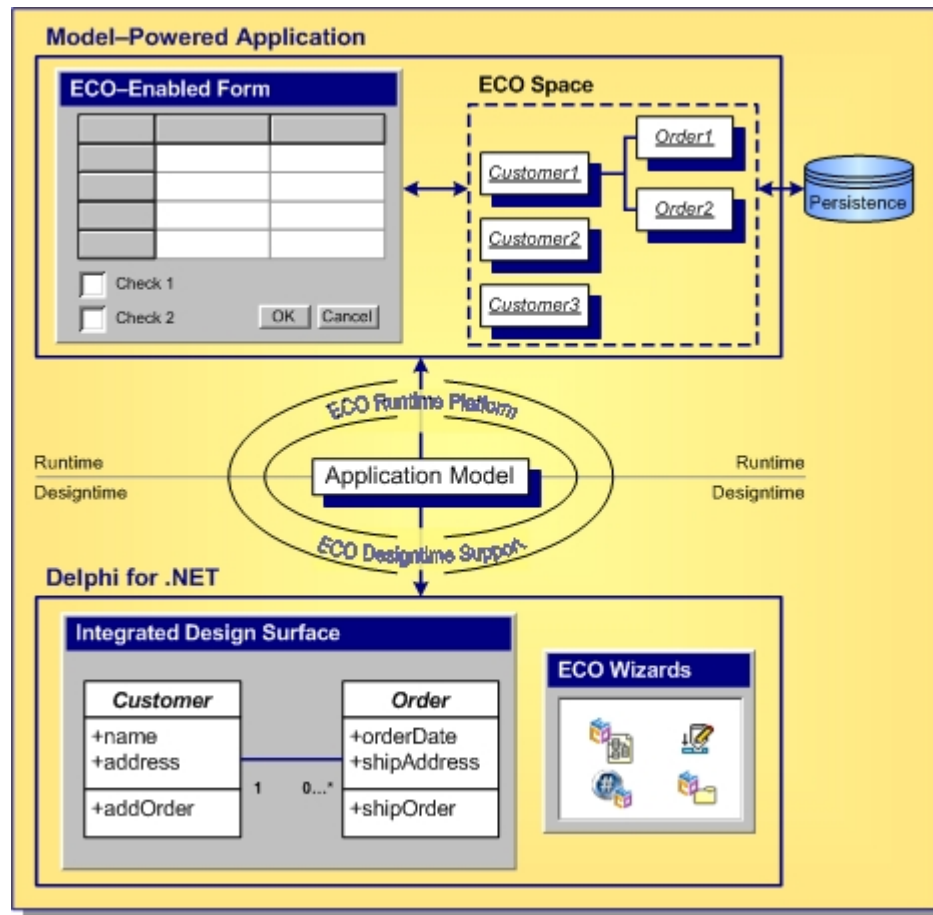
Modeling is a term used to describe the process of software design. Developing a model of a software system is roughly equivalent to an architect creating a set of blueprints for a large development project. Like a set of blueprints, a model not only depicts the system as a whole, it also allows you to focus in on specifics such as structural and behavioral details. Abstracted away from any particular programming language (and at some levels, even from specific technology), the model allows all participants in the development cycle to communicate in the same language. Borland's design driven architecture is an approach to software engineering where the modeling tools are completely integrated within the development environment itself. The software development process therefore centers primarily around the model, rather than the nuts and bolts of source code.

Central to Delphi 8 for .NET's modeling toolset is a powerful class library called the Enterprise Core Objects (ECO) framework. The ECO framework is both a design-time and a runtime platform that leverages the .NET Framework to put the system model at the center of your development activities.

The modeling tools in Delphi 8 for .NET are based on important industry standards published by the Object Management Group (OMG). These are, the *OMG Unified Modeling Language Specification*, the *Object Constraint Language Specification*, and the *OMG XML Metadata Interchange (XMI) Specification*. These specifications are freely available at the OMG website; see the link at the end of this topic for more information. This topic presents an overview of these concepts:

- Basic modeling and UML concepts.
- Introduction to the Enterprise Core Objects framework.
- Integrated modeling tools in the Delphi 8 for .NET IDE.

## Model-Powered Applications and the ECO Framework



The ECO framework is both a design-time and a run-time platform. The model is the central hub of activity for designers and developers; the ECO framework carries the design driven philosophy all the way through to the application end-users.

## Basic UML Concepts

Delphi 8 for .NET's modeling tools use UML notation to graphically depict the design of a software system. In the UML, different activities undertaken during design are represented by certain types of diagrams. For example, you would define the user roles and system requirements in a *use case* diagram, while installation requirements would be defined on a *deployment* diagram. Designers capture the vocabulary of the system they are modeling in *Static Structure*, or *Class* diagrams; this is the type of diagramming surface integrated in Delphi 8 for .NET.

**Note:** This topic covers Delphi 8 for .NET's built-in UML modeling tools, which are based entirely on the ECO framework. For information on code visualization, and producing a class diagram from an arbitrary set of Delphi source code, please refer to the Code Visualization link at the bottom of this topic.

## UML Packages

A UML package is a unit of organization analogous to a .NET namespace. Using packages, you can partition classes and other datatypes along just about any lines that make sense for your particular application. Usually classes are created along functional lines. For example, you could create a package to hold a set of user interface components. Like namespaces, UML packages can be nested within each other.

**Note:** While the concept of a UML package and a namespace are analogous, the UML packages you create with Delphi 8 for .NET are not implemented as .NET namespaces in source code. This is a major difference between the code visualization diagram, and the UML diagramming tools. When producing a code visualization diagram from an arbitrary Delphi source file, a .NET namespace is represented with a UML package symbol. On the code visualization diagram .NET namespaces are drawn in white, as opposed to ECO-enabled UML packages, which are drawn in yellow.

## UML Class Diagrams

The class diagram is both an architectural tool for overall system design, and an organizational tool for both designers and developers. The class diagram, when taken as a whole, allows designers and developers to work at a level of abstraction that is removed from the technical details of a particular platform or language. However, the individual UML elements that compose a class diagram were explicitly designed for the purpose of mapping the abstract system model onto language-specific types and idioms.

A class diagram focuses on the internal structure and the relationships of the classes in the model. The internal structure of a class consists of its attributes (including derived attributes), and operations. The kinds of relationships shown on the class diagram are inheritance (known as generalization in the UML), and links such as associations (including derived associations). You can also add OCL invariant constraints and query expressions to the model; you can call the ECO framework at runtime to enforce the constraints or to execute the query.

**Note:** As with .NET namespaces and UML packages, classes on the code visualization diagram are drawn in white, while ECO-enabled classes are drawn in yellow.

## The Object Constraint Language (OCL)

OCL is a side-effect free, pure expression language originally developed by IBM. Later, OCL was incorporated into the UML specification by OMG. OCL expressions return values, but they cannot be used to alter the model in any way. There are three aspects of working with OCL in Delphi 8 for .NET:

- **Setting invariant constraints.** OCL expressions are used at designtime to set restrictions on attributes and properties of classes. For example, you can specify that the Social Security attribute

of an `Employee` class cannot be set to null. You can then use the ECO framework to enforce this constraint at runtime.

- **Creating derived associations and attribute values.** Derived associations and attribute values are computed from other elements. OCL expressions are used to specify how this computation is carried out. Derived elements avoid redundancy, but perhaps even more important, they allow you to specify business rules in one place, rather than in multiple places in source code.
- **Navigating associations on the class diagram.** You can use OCL as a query language, to return collections of objects in your ECO Space. You can then use these collections directly in .NET data-aware controls to display and edit the values. For example, the following OCL statement `Person.allInstances.birthDate` retrieves a collection of the values of the `birthDate` attribute for all instances of the `Person` class. OCL's querying capabilities not only apply to class properties and attributes, but to associations you have defined on the class diagram. For example, say you have a `Person` class, and it has an association end called "Home", of type `Building`, and `Building` has an attribute called `Address`. When you evaluate the OCL expression `Home.Address` in the context of a specific `Person` object, you will get back the value of the `Address` attribute of the instance of `Home` associated with that `Person`.

## Introduction to the ECO Framework

---

Borland's approach to design driven application development is that models should be *implemented* and then *executed*, rather than interpreted by the developer. The difference between implementation and interpretation is that a precisely described model contains enough information that much of the source code needed to bring the model to life can be generated automatically, as opposed to being written by hand. Developers can now work at the level of the model, in the familiar context of Delphi classes.

Execution of the model means that the designtime support for creating the model carries through to runtime. A truly design driven software engineering process includes support not only for creation of a model, but also for maintaining and enforcing the integrity of the model at all phases of the application's lifetime. The common thread that runs through Delphi 8 for .NET's modeling toolset is the ECO framework. It is helpful to understand how the ECO framework splits its functionality into designtime support and runtime support.

### Designtime Support Features

- Persistence mechanism (RDBMS or XML file)
- Database configuration
- Schema creation/evolution
- Model validation
- OCL expression editor
- Object-relational mapping

### Runtime Support Features

- Undo/Redo mechanism
- OCL querying
- OCL evaluation
- Caching of objects
- Subscription mechanism
- Object versioning
- Transactions
- Binding to data-aware .NET UI controls

## ECO Spaces

An ECO Space is a container of objects. At runtime, the ECO Space contains actual instances of the classes in your model. It is helpful to think of the ECO Space as an actual instance of a model, much like an object is an instance of a class. The objects contained in the ECO Space retain the domain properties (attributes and operations) and relationships defined in the model.

As a container of objects, an ECO Space is both a cache, and a transactional context. Within the ECO Space, another component called a *persistence mapper* is used as the conduit between the persistence layer (either an RDBMS or XML file), and the instances of the classes defined in the model. When you configure an ECO Space using the IDE, you will select those UML packages in your model that you wish to persist in the ECO Space.

## The Borland.Eco Namespaces

This section contains a brief overview of the classes, interfaces, and other types that you will directly encounter in your source code when working with the ECO framework. Much of your interaction with the ECO framework will be through code automatically generated for you by the IDE. However, you will be working directly with the ECO framework when you need to access the runtime services it provides, such as the OCL evaluator, and the Undo/Redo mechanism.

## Borland.Eco.Services

The `Borland.Eco.Services` namespace is the primary interface to the runtime capabilities of the ECO framework. The services provided are:

- `IStateService`. Provides routines to determine if a given object or property has changed since the last database update.
- `IPersistenceService`. Provides routines for querying and saving, using the chosen persistence mechanism of the ECO Space.
- `IDirtyListService`. Provides routines for returning lists of objects that have changed since the last database update.
- `IExtentService`. Queries the ECO Space for all objects of a given class.

- **IObjectFactoryService.** Creates a new instance of a given class using the model's UML type system. When creating new instances, you can use either the **IObjectFactoryService** interface, or you can use the constructors defined on your classes.
- **IOclService.** Provides routines for evaluating OCL expressions and returning the results.
- **IVersionService.** Provides routines for creating and storing versions of objects in the ECO Space. Versioned objects are tagged with version numbers starting with 1. Versioned objects support multiple, read-only instances, and one "live" instance. The latest version, tagged with the special version number **CurrentVersion**, is always the live, writeable instance.
- **IUndoService.** Provides routines for supporting Undo/Redo functionality on objects in the ECO Space.

These services are available using methods on the subclass of **EcoSpace** that the IDE generates for you when you start a new ECO application (or when you add a new **EcoSpace** subclass to an existing application).

## Borland.Eco.Handles

The **Borland.Eco.Handles** namespace contains the components used for defining and interacting with ECO Spaces. The components in this namespace fall into three categories:

- The **DefaultEcoSpace** class
- The **OclVariables** class
- The **ElementHandle** abstract class

You cannot instantiate a **DefaultEcoSpace** object directly; instead the IDE will generate a subclass for you when you use either the *ECO Application wizard*, or the *ECO Space wizard*.

The **ElementHandle** class is the abstract superclass of all model elements. **ElementHandle** objects represent values. The handle might represent a single value, or a collection of values. Subclasses of **ElementHandle** implement the **IListSource** interface, making them suitable for use as data sources in the .NET databinding architecture.

The **OclVariables** class allows you to create variables for use in OCL expressions. These variables are then bound to, and reflect the value of the actual element handles in the ECO Space.

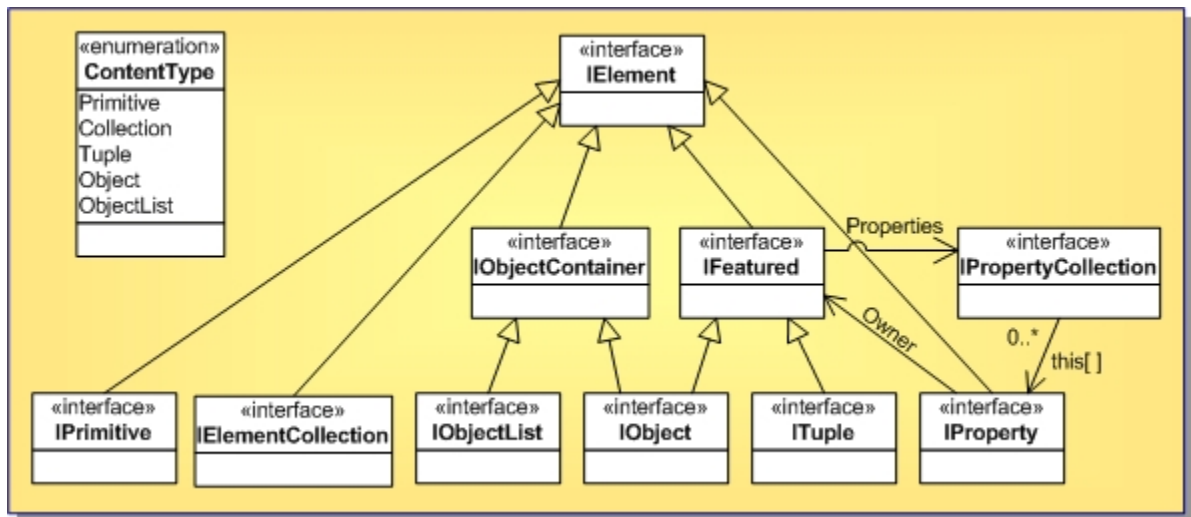
## Borland.Eco.ObjectRepresentation

There are two ways to access the objects in your application's ECO Space. The most direct way is to simply access the class' properties and methods through the source code generated from your class diagrams. The **Borland.Eco.ObjectRepresentation** namespace provides a second, more generic way that does not involve direct use of the types defined in the generated source code.

Instead, access to the objects is accomplished through a set of interfaces that all ECO classes implement indirectly (i.e. they are implemented for you by the class diagramming and code generation tools). As the name **ObjectRepresentation** implies, these interfaces expose the model the way it is represented internally within the ECO Space. One reason for accessing objects through the

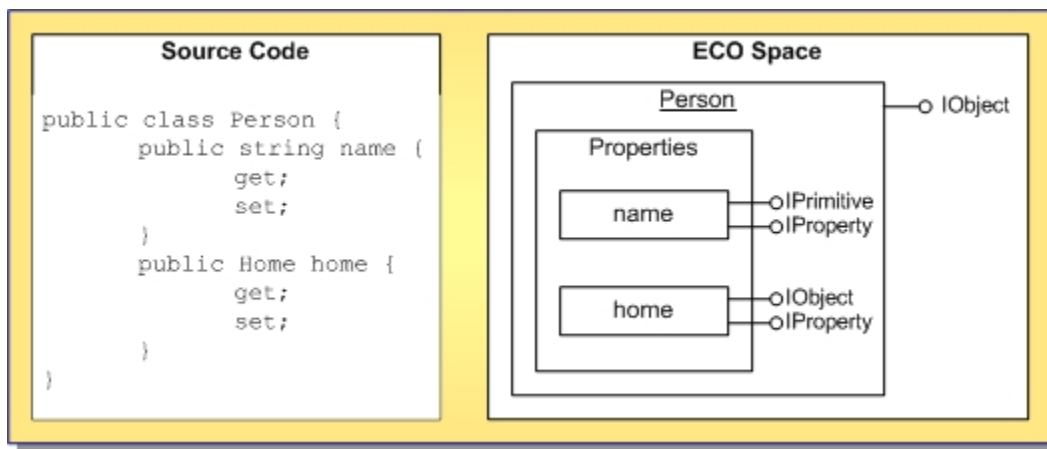
`ObjectRepresentation` interfaces is if you are creating user interface controls that are designed to work with objects in an arbitrary ECO Space.

The second, more common use of the `ObjectRepresentation` interfaces is when you work with ECO services such as persistence, Undo/Redo, object versioning, and OCL evaluation; these services are defined in the `Borland.Eco.Services` namespace, and are available through your application's ECO Space object. The `Services` APIs take the `ObjectRepresentation` interfaces as parameters, and return references to them, which you can then use to call methods on the interface, or to cast to a type defined in your model. The following diagram shows the interfaces defined in the `ObjectRepresentation` namespace.



The root interface, `IElement`, contains a property called `ContentType` that you can examine to determine how to cast the interface reference. `IElement` represents all runtime elements, including the objects themselves, their attributes, the associations between classes in your model, and primitive types such as strings and collections. The key to linking your model as it exists on the class diagrams with its representation within the ECO framework, is to think of a single class as a generic container of elements. These elements might be primitive types, or objects, or they might be collections themselves, which in turn contain more elements.

For example, suppose you had a `Person` class in your model. You can think of this class as a container for a set of properties, such as `personName`, `home`, and `ownedBuildings`. The property `personName` is a string (a primitive type), the property `home` is an object of another class (which has its own set of properties), and `ownedBuildings` is a collection of objects. The following diagram shows how the mapping is made from the source code declaration to the interfaces implemented by the object's representation within the ECO Space.



## Borland.Eco.Subscription

The `Borland.Eco.Subscription` namespace contains interfaces and classes related to the ECO framework's subscription service. With subscriptions, you can tell the framework to send a notification when a particular event occurs within the object space. For example, you can have the framework send a notification when a certain object (or more likely, a collection of objects) within the ECO Space changes. The interfaces and classes provided in this namespace are:

- `ISubscriber`. You must implement this interface on all classes that you want to act as subscribers.
- `SubscriberAdapterBase`. This is an abstract base class you can use where you don't want to implement the `ISubscriber` interface directly.

Typically you will use the `SubscriberAdapterBase` class, overriding its `DoReceive` method, as shown in the following example:

```
uses
    Borland.Eco.Subscription,
    Borland.Eco.Services;

type
    TMySubscribingClass = class
    private
        type
            TMySubscriberAdapter = class(SubscriberAdapterBase)
            strict protected
                procedure DoReceive(sender: System.Object; e: EventArgs); override;
            end;
        private
            FmyAdapter: TMySubscribingClass.TMySubscriberAdapter;
            FextentService: IExtentService;
            procedure ExtentServiceChanged;
            procedure RespondToEvent;
        end;
```



```

procedure TMySubscribingClass.TMySubscriberAdapter.DoReceive(sender:
System.Object; e: EventArgs);
begin
    // ActualSubscriber is a property of SubscriberAdapterBase.
    (ActualSubscriber as TMySubscribingClass).RespondToEvent;
end;

procedure TMySubscribingClass.ExtentServiceChanged;
begin
    // Drop old subscriptions if any...
    if FmyAdapter <> nil then
        FmyAdapter.Deactivate();
        FmyAdapter := TMySubscriberAdapter.Create(self);
        // Place a subscription
        FextentService.SubscribeToObjectAdded(FmyAdapter, nil);
end;

procedure TMySubscribingClass.RespondToEvent;
begin
    // Add code to handle the event
end;

```

## Borland.Eco.Persistence

The `Borland.Eco.Persistence` namespace contains classes related to saving objects in an ECO Space out to disk. The ECO framework supports persistence to either a relational database, or an XML file. There are three components of primary interest in this namespace:

- `PersistenceMapperBdp`. This component is used for saving objects to a relational database using the Borland Data Provider classes for database connectivity.
- `PersistenceMapperSqlServer`. This component uses the `SqlConnection` component for database connectivity.
- `PersistenceMapperXML`. This component is used to store objects in an XML file.

These components are used by dropping them onto the *ECO Space designer*, and then connecting them to the `PersistenceMapper` property of the application's ECO Space.

In addition, the `Borland.Eco.Persistence` namespace contains three interfaces that are used to map a single class attribute to one or more database columns. These interfaces are:

- `IAttributeMapping`
- `ISingleColumnAttributeMapping`
- `INonBooleanBooleanMapping`

The namespaces nested within `Borland.Eco.Persistence` contain the classes that implement the default attribute mappings for the supported databases.

## Borland.Eco.UmlRt

The `Borland.Eco.UmlRt` namespace implements a subset of the foundation package in the UML metamodel version 1.4. The interfaces in this namespace are used to access the UML elements that comprise the model's type system. Data pertaining to the types within the type system are accessed through the interfaces in the `UmlRt` namespace. For example, this namespace contains the interfaces `IClassifier`, `IClass`, and `IAttribute`; these interfaces can be used to access the type system's metadata.

The top-level interface is `IEcoTypeSystem`. There are two ways to access the type system:

- The `TypeSystem` property of the `EcoSpace` class.
- Through the `GetEcoService` method, available on classes that implement the `IEcoServiceProvider` interface.

Once you have the `IEcoTypeSystem`, you can examine its properties, such as `AllClasses`, which returns a collection of metadata on all of the UML classes defined in the model.

## Integrated Modeling Tools in Delphi 8 for .NET

---

Through the ECO framework and integrated Together UML technologies, the Delphi 8 for .NET IDE provides a set of modeling tools that allow the creation of class diagrams, automatic source code generation, database schema generation and evolution, and more.

## ECO Projects and Wizards

The IDE has four code-generating wizards to help you develop an ECO-enabled application:

- **ECO Application.** Creates an application with a default ECO space, a root UML package, and an ECO enabled Windows form.
- **ECO Enabled Windows Form.** Adds an ECO enabled Windows form to your project.
- **ECO Space.** Creates a new subclass of `DefaultEcoSpace` in your project.
- **ECO UML Package.** Adds a new package to the root UML package for your project. You can also add new UML packages directly from the class diagram.



Code generated by these wizards will include all of the necessary ECO-related .NET attributes and default interface implementations.



## The Model View Window

Unlike the *Project Manager*, the *Model View* window lets you navigate your project based on the logical relationships between the classes and other elements in source code. The *Model View* window actually presents these relationships from two very different perspectives: code visualization, which derives a UML class diagram from arbitrary Delphi code, and visualization of ECO-enabled packages and classes. Code visualization scans source code and derives the elements, such as namespaces and classes, and the relationships between them. Therefore, you can think of code visualization as a superset, or raw

view of the logical relationships in your code. Because it gives you an unfiltered view by design, code visualization will naturally expose some implementation details behind the ECO framework.

A notable example of this is the fact that UML packages are actually implemented as classes, as opposed to .NET namespaces as one might expect. On a code visualization diagram, you will see UML packages represented as classes within your project's namespace. On an ECO class diagram however, you will see the true, logical representation of the UML package. When developing ECO applications, a general rule to follow is to use code visualization diagrams to view the non-ECO elements of your project.

In the *Model View* window, all of your ECO-enabled UML packages and classes will be grouped under a top-level root package in the project tree. The default name of the root UML package is `CoreClassesPackage`. The root UML package node (and all UML packages underneath it) is distinguished from an ordinary .NET namespace node by its icon. The icon, , represents a .NET namespace discovered by code visualization. The icon, , represents a UML package.

Similarly, ECO classes are distinguished from their code visualization counterparts by a different icon. The icon, , represents a class discovered by code visualization. The icon, , represents an ECO-enabled class.

## The Class Diagram

The integrated UML diagramming tools support the following activities:

- Creating UML packages
- Creating classes
- Drawing generalization (inheritance) links between classes
- Drawing associations between classes
- Attaching notes to diagram elements
- Adding attributes and operations to classes

The class diagram itself is another type of designer surface. You can add new UML elements to the diagram, including associations and notes, using the *Tool Palette*. You can select UML elements on the diagram and set their properties in the *Object Inspector*. As you work on the class diagram, Delphi 8 for .NET generates the ECO-enabled source code that implements the model.

Class diagrams are opened from the *Model View window*. Each UML package you create has its own primary class diagram, and this diagram cannot be deleted (it can be renamed, however). The class diagram nodes are grouped underneath their UML package in the *Model View tree*. The primary class diagram for a UML package always shows the entire contents of the package; it displays all of the sub-packages, classes, and relationships that exist within that package. When you add a new element to a UML package it is automatically represented on the primary class diagram.

You can also create secondary class diagrams within a UML package, if you want to show a subset of the classes within the package. Unlike with the primary class diagram, new elements you add to the

package are not automatically added to secondary diagrams. Secondary diagrams can be renamed and deleted as needed.

Any UML elements you add to a primary or secondary class diagram will be contained within the UML package that owns the diagram. To show elements in other UML packages, you must create a *shortcut* to the element. You can do this through the context menu of the class diagram. Shortcuts are displayed on the diagram with a small arrow icon in their lower left corner. Once a shortcut has been created, you can add associations between it and the classes in the UML package that owns the diagram.

# Working with Unmanaged Code

---

Borland's Delphi 8 for .NET provides the capability to work with the .NET features that support unmanaged code. If you have COM or ActiveX components that you want to use within the .NET framework, you can use the .NET COM Interop capabilities from within Delphi 8 for .NET while building your applications.

# Using COM Interop in Managed Applications

---

COM Interop is a .NET service that allows seamless interoperability between managed and unmanaged code. The COM Interop service is a two-way bridge: It allows you to leverage existing COM servers and ActiveX Controls in new .NET applications, as well as to expose .NET components in legacy, unmanaged applications.

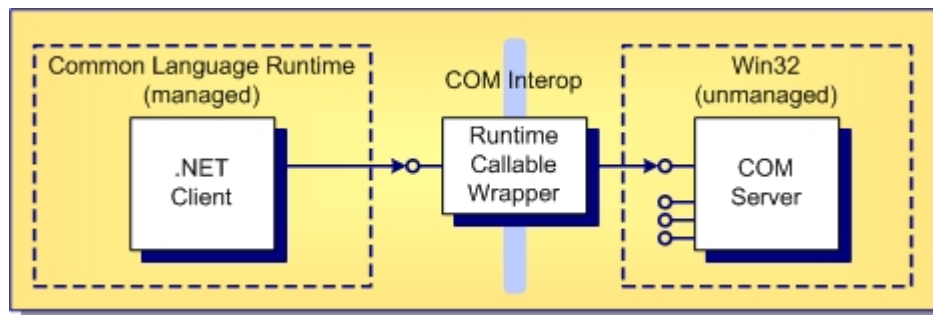
The Delphi 8 for .NET IDE features tools that will help you integrate your legacy COM servers and ActiveX Controls into managed applications. Within the IDE, you can add references to unmanaged DLLs to your project, and then browse the types contained in them, just as you can with managed assemblies. You can add ActiveX Controls to the *Tool Palette*, and then drop them on your forms as you would with any .NET component.

The following topics are covered in this overview:

- Introduction to the terminology of COM Interop. If you are already familiar with these concepts, you can skip directly to the section on Delphi 8 for .NET IDE features and tools for COM/Interop.
- Introduction to some of the .NET Framework SDK tools for working with COM/Interop.
- Using COM Interop Assemblies in the IDE.
- Deploying applications that use COM Interop.
- Summary

## COM Interop Overview

---



Seamless interoperability is achieved through stand-in objects called Runtime Callable Wrappers (RCW). The RCW is a layer of communication between your managed application, and the actual unmanaged COM server.

## COM Interop Terminology

---

The .NET Framework has a rich collection of terms and three-letter acronyms. This section will help you understand the terminology you will encounter when reading other COM Interop literature.

### Metadata

In the context of .NET and COM, metadata is a term used to mean type information. In COM, type information can be stored in a variety of ways. For instance, a C++ header file is a language-specific container for type information. A type library is also a container for type information, but being a binary format, type libraries are language neutral. Unlike the COM development model where type libraries are not required, language neutral metadata is mandatory for all .NET assemblies. Every assembly is self-describing; its metadata contains complete type information, including private types and private class members.

### Custom Attributes

Developers often tag program entities (such as classes and their methods) with descriptive attributes such as static, private, protected, and public. In the .NET Framework, you can tag any entity, including classes, properties, methods, and even assemblies themselves, with an attribute of your own design and meaning. Custom attributes are expressed in source code, and are processed by the compiler. At the end of the build process, custom attributes are emitted into the output assembly just like all metadata.

### Reflection

A unique characteristic of the .NET Framework is that type information is not lost during the compilation process. Instead, all metadata, including custom attributes, is emitted by the compiler into the final output assembly. Metadata is available at runtime, through .NET Reflection services. The .NET Framework SDK provides a reflection tool called ildasm that allows the developer to open any .NET assembly, and inspect the types declared therein. Such reflection tools often allow programmers to directly view the IL code generated by the compiler. The Delphi 8 for .NET IDE contains its own integrated reflection tool, in the form of the meta data explorer tool that appears when you open a .NET assembly.

### Global Assembly Cache

In COM, components can be deployed anywhere on the user's machine. Usually, a component's installation script records its location in the system registry. Command-line tools such as regsvr32 and tregsvr can also add and remove COM components from the registry. Registration of components is required in COM programming, even if the components are not intended to be shared by multiple applications.

The .NET programming model drastically simplifies deployment of applications and components. On the .NET platform, non-shared components are deployed directly into the application's local installation directory; no registration is required. Alternatively, a non-shared component can be deployed in a

directory specified in the application's configuration file. Again, registration is not required for this deployment scenario.

Shared components are installed into a special location called the Global Assembly Cache (GAC). The GAC is an evolution of the system registry (though it is a completely separate mechanism and is not associated with the registry at all). The GAC exists in the file system in a folder called \Windows\Assembly. The .NET Framework supports simultaneous, or "side-by-side" deployment of different versions of the same component. When you view the Global Assembly Cache folder using Windows Explorer, you are actually looking at the GAC through a special shell extension. The shell extension presents all of the assemblies that have been installed into the GAC, with their version, culture, and public key information.

There are three ways to install a .NET component into the GAC. The first way is to use the Framework SDK command-line tool called `gacutil`, which is discussed below. Another way is to install a component into the GAC is to navigate to the \Windows\Assembly folder using Windows Explorer, and then simply drag and drop the assembly into the directory listing pane. Finally, you can also use the .NET Configuration management tool, which is accessible through the Windows Control Panel.

## Strong Names

The concept of a strong name is similar to that of the 128-bit Globally Unique Identifier (GUID) in COM programming. A GUID is a name that is guaranteed to be globally unique. Every .NET assembly has a basic name, which consists of a text string, a version number, and optional culture information. For shared assemblies installed into the GAC, the basic name alone is not enough to guarantee the assembly is uniquely identified. To generate a globally unique name, an encryption key with public and private components is used to generate a digital signature. The signature is then applied to the assembly using the .NET Framework SDK Assembly Linker (`al.exe`), or by using assembly attributes in source code.

## Runtime Callable Wrappers and COM Callable Wrappers

Accessing a component, be it a .NET component or a COM server, is largely transparent. That is, when you are using a COM server in a .NET application, the COM server looks like any other .NET component. Similarly a .NET component, when exposed to an unmanaged application through COM Interop, looks like a COM server. This transparency is accomplished by behind-the-scenes proxies, or wrapper objects.

When you use a COM object in a managed application, the Common Language Runtime (CLR) creates an RCW, which is the interface between managed and unmanaged code. The complexities of data marshaling and reference counting are handled by the RCW. In fact the RCW does not even expose the *IUnknown* and *IDispatch* interfaces.

When you use a .NET component in an unmanaged application, the system creates a stand-in called a COM Callable Wrapper (CCW).



## Primary Interop Assembly

In the COM programming model, once a GUID is assigned to a type, the GUID always refers to that specific type no matter where the type appears. For example, a common interface might be defined in many different type libraries, but each separate type library would have to define the interface with the same GUID, so the duplication is not a problem. However, if you generate COM Interop assemblies for these separate type libraries, a new and distinct assembly would be created for each type library. Each of these separate assemblies would contain distinct types (as far as the CLR is concerned). The strong identity and self-describing nature of .NET assemblies is actually working against you in this case. Here, it is leading to a GAC that is cluttered with interop assemblies that all contain RCWs for the same type library. Worse yet, to the CLR each assembly contains distinct and incompatible types, because each one has a different strong name.

To avoid this proliferation of assemblies and potential type incompatibilities, the framework gives you the ability to designate one assembly as the primary interop assembly for a type library. A primary interop assembly is always signed with a strong name, by the original publisher of the type library.

## COM Interop Tools in the .NET Framework SDK

---

Some of the functionality provided by the .NET Framework SDK tools is exposed in the development environment. This section is not intended to be a complete reference for these tools; it is merely a starting point for more exploration of the .NET Framework SDK, and hopefully will give you a bit more understanding of how to work with COM Interop technology in the IDE.

### Importing and Exporting Type Libraries

Tlbimp is a command-line tool that you can use to generate a .NET assembly from a type library. Tlbimp will operate on a type library directly, or on an unmanaged DLL that contains a type library as an embedded resource. Note the assembly produced by tlbimp contains code for only the RCW, not for the original COM object itself. Therefore you must still deploy and register the COM object on the end-user's machine. The assembly also contains the types described in the type library, expressed as metadata. Tlbimp uses a command line switch to produce a primary interop assembly.

The .NET Framework SDK contains another command-line tool called tlbexp that is used to create a type library from a .NET assembly. Such an exported type library would then be used to expose the .NET component as a COM server, for use within an unmanaged application.

### Importing ActiveX Control Libraries

Aximp is a command-line tool that generates an ActiveX Control wrapper assembly. This assembly is required so that the ActiveX Control can be used on a Windows Form. A special utility is required, because a Windows Form can only host controls that are derived from the System.Windows.Forms.Control class, and the tlbimp utility does not create a wrapper derived from that class.

The aximp tool will generate both interop assemblies (as with tlbimp, this includes dependent assemblies), and the ActiveX wrapper assembly. Like tlbimp, aximp has command-line switches to sign

the assemblies produced with a strong name. Unlike `tlbimp`, `aximp` cannot generate a primary interop assembly.

## Generating Strong Names

If you are deploying a .NET component into the GAC, you will need to sign your assembly with a strong name key. This is done by using a .NET Framework SDK command-line tool called `sn`. The assembly is signed with the strong name in one of three ways:

- By specifying the strong name key file in the assembly linker (`al`) command line
- By tagging the assembly with the `AssemblyKeyFile` attribute
- By using a technique called "delay signing"

When using delay signing, the assembly is signed with the public portion of the key file at build time. Before shipping the assembly, the `sn` tool is used again to sign the assembly with the private key.

## Deploying a .NET Component to the Global Assembly Cache

The .NET Framework SDK utility called `gacutil` is a command-line program that is used to install, remove, and view components in the GAC. The `gacutil` command is usable from installation scripts as well as from batch files. The `gacutil` command supports installation and removal of shared assemblies, with and without the use of reference counting. It is recommended that the non-reference counted command switches be used only during development. Installation scripts that use `gacutil` to install shared components should always use the reference counted command line switches.

## Using COM Interop Assemblies in the IDE

---

All of the functionality encompassed by the .NET Framework SDK command-line tools is in fact exposed by the .NET Framework Class Library itself. The Delphi 8 for .NET IDE also takes advantage of the .NET Framework classes to expose interoperability features. The IDE goes beyond the capabilities of the command-line tools, however, making interoperation with unmanaged components even easier.

## Type Libraries and Interop Assemblies

The IDE initiates the creation of interop assemblies through the *Project Manager*. When you add a reference to a DLL to your project, you can select from registered type libraries and unmanaged DLLs, or you can browse to an unregistered component.

The IDE creates one interop assembly for each imported type library or DLL. The assemblies are named `Interop.LibraryName.dll`, where *LibraryName* is the name of the type library. The name of the library is specified in the library statement in IDL source code, so the file name of the generated assembly might be different from that of the original DLL or type library. Each interop assembly (and all of its dependent assemblies) are added to your project as referenced assemblies. The types contained in the interop

assembly are added to a namespace with the same name as the type library; Again, this is derived from the library statement in IDL source code.

If the assembly you reference has a primary interop assembly, the IDE will recognize this and avoid generating a new interop assembly. In this case, the IDE will add a reference to the primary interop assembly in the GAC, and it will not copy the assembly to your local project directory.

## Importing ActiveX Controls

To use an ActiveX Control in your managed application, you must first add the control to the tool palette. This will create both an interop assembly, and an ActiveX assembly with a wrapper class derived from `System.Windows.Forms.AxHost`. The ActiveX wrapper assembly will be named `AxInterop.LibraryName.dll`, where *LibraryName* is the name of the type library. Dragging the control from the palette onto a Windows Form will automatically add references to both assemblies to your project.

Once on your form, the ActiveX Control can be treated as any other .NET component. You can select the control, and set its properties and event handlers in the *Object Inspector*. The ActiveX Control wrapper will expose the properties of the `Windows.Forms.Control` class, while properties exposed by the ActiveX Control will be grouped under the *Misc* category.

## Interop Assemblies and the Project Manager

Interop assemblies (including ActiveX Control wrapper assemblies) generated by the IDE are kept in a separate folder called `COMImports`, underneath your project. Each generated assembly will have its 'Copy Local' property set, meaning that when the project is built, the assembly will be copied to the folder where the final build target of the project is kept. The exceptions to this rule are primary interop assemblies, which are deployed in the GAC. When you add a reference to a primary interop assembly, the IDE will not copy the assembly to the `COMImports` folder. The assembly will still be shown in the *Project Manager*, however, if you right click on it to display its properties, you will notice that the 'Copy Local' setting is turned off.

The list of referenced assemblies (including those that are not interop assemblies) is an attribute of your project. If the `COMImports` folder (or one of the interop assemblies contained therein) does not exist when you open a project, the IDE will attempt to recreate it. If the IDE cannot create an interop assembly, it will still be shown as a referenced assembly in the *Project Manager*; the IDE will highlight such an assembly so that you know it currently does not exist (or is not registered) on the machine.

## Deploying Applications That Use COM Interop

---

Two things are important to keep in mind when working with unmanaged components. First, remember that an interop assembly is not a replacement for the COM server; it is a stand-in, or proxy for it. The interop assemblies produced by `tlbimp` and Delphi 8 for .NET are not transformations of the component's unmanaged code into managed code. Every file required by the component in an unmanaged deployment environment, must also be deployed in a managed environment *in addition to* the interop assemblies. Second, the .NET Framework's interop services do not circumvent the requirement of

registering the COM server on the end-user's machine. Note the registration requirement also applies during the development of your managed application.

As with any other .NET assembly, an interop assembly can be deployed alongside the managed executable in the installation folder, or it can be deployed in the GAC. If you deploy the interop assembly into the GAC, you must give it a strong name during development. Primary interop assemblies are always deployed into the GAC; however, just because an assembly is deployed to the GAC, does not automatically make it a primary interop assembly. An interop assembly is designated as a primary interop assembly by using the /primary command-line option of the `tlbimp` utility. The IDE currently has no built-in support for creating primary interop assemblies. Unmanaged COM servers can be deployed anywhere on the end-user's machine, however, as noted previously, you must still register unmanaged components when your application is installed.

# Using Platform Invoke with Delphi 8 for .NET

---

This topic describes the basic techniques of using unmanaged APIs from Delphi 8 for .NET. Some of the common mistakes and pitfalls are pointed out, and a quick reference for translating Delphi data types is provided. This topic does not attempt to explain the basics of platform invoke or marshaling data. Please refer to the links at the end of this topic for more information on platform invoke and marshaling. Understanding attributes and how they are used is also highly recommended before reading this document.

The Win32 API is used for several examples. For further details on the API functions mentioned, please see the Windows Platform SDK documentation.

The following topics are discussed in this section:

- Calling unmanaged functions
- Structures
- Callback functions
- Passing Object References
- Using COM Interfaces

## Calling Unmanaged Functions

---

When calling unmanaged functions, a managed declaration of the function must be created that represents the unmanaged types. In many cases functions take pointers to data that can be of variable types. One example of such a function is the Win32 API function [SystemParametersInfo](#) that is declared as follows:

```
BOOL SystemParametersInfo(  
    UINT uiAction,    // system parameter to retrieve or set  
    UINT uiParam,     // depends on action to be taken  
    PVOID pvParam,    // depends on action to be taken  
    UINT fWinIni      // user profile update option  
);
```

Depending on the value of [uiAction](#), [pvParam](#) can be one of dozens of different structures or simple data types. Since there is no way to represent this with one single managed declaration, multiple overloaded versions of the function must be declared (see [Borland.Vcl.Windows.pas](#)), where each overload covers one specific case. The parameter [pvParam](#) can also be given the generic declaration [IntPtr](#). This places the burden of marshaling on the caller, rather than the built in marshaler. Note that the data types used in a managed declaration of an unmanaged function must be types that the default marshaler supports. Otherwise, the caller must declare the parameter as [IntPtr](#) and be responsible for marshaling the data.

## Data Types

Most data types do not need to be changed, except for pointer and string types. The following table shows commonly used data types, and how to translate them for managed code:

Unmanaged Data Type	Managed Data Type	
	Input Parameter	Output Parameter
Pointer to string (PChar)	String	StringBuilder
Untyped parameter/buffer	TBytes	TBytes
Pointer to structure (PRect)	const TRect	var TRect
Pointer to simple type (PByte)	const Byte	var Byte
Pointer to array (PInteger)	array of Integer	array of Integer
Pointer to pointer type (^PInteger)IntPtr		IntPtr

IntPtr can also represent all pointer and string types, in which case you need to manually marshal data using the Marshal class. When working with functions that receive a text buffer, the StringBuilder class provides the easiest solution. The following example shows how to use a StringBuilder to receive a text buffer:

```
function GetText(Window: HWND; BufSize: Integer = 1024): string;
var
    Buffer: StringBuilder;
begin
    Buffer := StringBuilder.Create(BufSize);
    GetWindowText(Window, Buffer, Buffer.Capacity);
    Result := Buffer.ToString;
end;
```

The StringBuilder class is automatically marshaled into an unmanaged buffer and back. In some cases it may not be practical, or possible, to use a StringBuilder. The following examples show how to marshal data to send and retrieve strings using [SendMessage](#):

```
procedure SetText(Window: HWND; Text: string);
var
    Buffer: IntPtr;
begin
    Buffer := Marshal.StringToHGlobalAuto(Text);
    try
        Result := SendMessage(Window, WM_SETTEXT, 0, Buffer);
    finally
        Marshal.FreeHGlobal(Buffer);
    end;
```

```
end;  
end;
```

An unmanaged buffer is allocated, and the string copied into it by calling `StringToHGlobalAuto`. The buffer must be freed once it's no longer needed. To marshal a pointer to a structure, use the `Marshal.StructureToPtr` method to copy the contents of the structure into the unmanaged memory buffer.

The following example shows how to receive a text buffer and marshal the data into a string:

```
function GetText(Window: HWND; BufSize: Integer = 1024): string;  
var  
    Buffer: IntPtr;  
begin  
    Buffer := Marshal.AllocHGlobal(BufSize * Marshal.SystemDefaultCharSize);  
    try  
        SendMessage(Window, WM_GETTEXT, BufSize, Buffer);  
        Result := Marshal.PtrToStringAuto(Buffer);  
    finally  
        Marshal.FreeHGlobal(Buffer);  
    end;  
end;
```

It is important to ensure the buffer is large enough, and by using the `SystemDefaultCharSize` method, the buffer is guaranteed to hold `BufSize` characters on any system.

## Advanced Techniques

When working with unmanaged API's, it is common to pass parameters as either a pointer to something, or `NULL`. Since the managed API translations don't use pointer types, it might be necessary to create an additional overloaded version of the function with the parameter that can be `NULL` declared as `IntPtr`.

## Special Cases

There are cases where a `StringBuilder` and even the `Marshal` class will be unable to correctly handle the data that needs to be passed to an unmanaged function. An example of such a case is when the string you need to pass, or receive, contains multiple strings separated by `NULL` characters. Since the default marshaler will consider the first `NULL` to be the end of the string, the data will be truncated (this also applies to the `StringToHGlobalXXX` and `PtrToStringXXX` methods). In this situation `TBytes` can be used (using the `PlatformStringOf` and `PlatformBytesOf` functions in `Borland.Delphi.System` to convert the byte array to/from a string). Note that these utility functions do not add or remove terminating `NULL` characters.

When working with COM interfaces, the `UnmanagedType` enumeration (used by the `MarshalAsAttribute` class) has a special value, `LPStruct`. This is only valid in combination with a `System.Guid` class, causing the marshaler to convert the parameter into a Win32 GUID structure. The function `CoCreateInstance` that is declared in Delphi 7 as:

```
function CoCreateInstance([MarshalAs(UnmanagedType.LPStruct)] clsid:
TCLSID;
    [MarshalAs(UnmanagedType.IUnknown)] unkOuter: TObject;
    dwClsContext: Longint;
    [MarshalAs(UnmanagedType.LPStruct)] iid: TIID;
    [MarshalAs(UnmanagedType.Interface)] out pv
): HRESULT;
```

This is currently the only documented use for `MarshalAs(UnmanagedType.LPStruct)`.

## Structures

The biggest difference between calling unmanaged functions and passing structures to unmanaged functions is that the default marshaler has some major restrictions when working with structures. The most important are that dynamic arrays, arrays of structures and the `StringBuilder` class cannot be used in structures. For these cases `IntPtr` is required (although in some cases string paired with various marshaling attributes can be used for strings).

## Data Types

The following table shows commonly used data types, and how to “translate” them for managed code:

Unmanaged Data Type	Managed Data Type	
	Input Parameter	Output Parameter
Pointer to string (PChar)	String	IntPtr
Character array ( <code>array[a..b] of Char</code> )	String	String
Array of value type ( <code>array[a..b] of Byte</code> )	<code>array[a..b] of Byte</code>	<code>array[a..b] of Byte</code>
Dynamic array ( <code>array[0..0] of type</code> )	IntPtr	IntPtr
Array of struct ( <code>array[1..2] of TRect</code> )	IntPtr or flatten	IntPtr or flatten
Pointer to structure (PRect)	IntPtr	IntPtr
Pointer to simple type (PByte)	IntPtr	IntPtr
Pointer to array (PInteger)	IntPtr	IntPtr
Pointer to pointer type (^PInteger)	IntPtr	IntPtr

When working with arrays and strings in structures, the `MarshalAs` attribute is used to describe additional information to the default marshaler about the data type. A record declared in Delphi 7, for example:

```
type
    TMyRecord = record
```



```

    IntBuffer: array[0..31] of Integer;
    CharBuffer: array[0..127] of Char;
    lpszInput: LPTSTR;
    lpszOutput: LPTSTR;
end;

```

Would be declared as follows in Delphi 8 for .NET:

```

type
  [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
  TMyRecord = record
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 32)]
    IntBuffer: array[0..31] of Integer;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    CharBuffer: string;
    [MarshalAs(UnmanagedType.LPTStr)]
    lpszInput: string;
    lpszOutput: IntPtr;
end;

```

The above declarations assume that the strings contain platform dependant TChar's (as commonly used by the Win32 API). It is important to note that in order to receive text in `lpszOutput`, the Marshal.AllocHGlobal method needs to be called before passing the structure to an API function.

A structure can contain structures, but not pointers to structures. For such cases an IntPtr must be declared, and the Marshal.StructureToPtr method used to move data from the managed structure into unmanaged memory. Note that StructureToPtr does not allocate the memory needed (this must be done separately). Be sure to use Marshal.SizeOf to determine the amount of memory required, as Delphi's SizeOf is not aware of the MarshalAs attribute (in the example above, `CharBuffer` would be 4 bytes using Delphi's SizeOf when it in fact should occupies 128 bytes on a single byte system). The following examples show how to send messages that pass pointers to a structure:

```

procedure SetRect(Handle: HWND; const Rect: TRect);
var
    Buffer: IntPtr;
begin
    Buffer := Marshal.AllocHGlobal(Marshal.SizeOf(KindOf(TRect)));
    try
        Marshal.StructureToPtr(TObject(Rect), Buffer, False);
        SendMessage(Handle, EM_SETRECT, 0, Buffer);
    finally
        Marshal.DestroyStructure(Buffer, KindOf(TRect));
    end;
end;

procedure GetRect(Handle: HWND; var Rect: TRect);
var

```

```

    Buffer: IntPtr;
begin
    Buffer := Marshal.AllocHGlobal(Marshal.SizeOf(Kind(TRect)));
    try
        SendMessage(Handle, EM_GETRECT, 0, Buffer);
        Rect := TRect(Marshal.PtrToStructure(Buffer, Kind(TRect)));
    finally
        Marshal.DestroyStructure(Buffer, Kind(TRect));
    end;
end;

```

It is important to call `DestroyStructure` rather than `FreeHGlobal` if the structure contains fields where the marshaling layer needs to free additional buffers (see the documentation for `DestroyStructure` for more details).

## Advanced topics

Working with unmanaged API's it is not uncommon to need to convert a byte array into a structure (or retrieve one or more fields from a structure held in a byte array), or vice versa. Although the `Marshal` class contains a method to retrieve the offset of a given field, it is extremely slow and should be avoided in most situations. Informal performance tests show that for a structure with eight or nine numeric fields, it is much faster to allocate a block of unmanaged memory, copy the byte array to the unmanaged memory and call `PtrToStructure` than finding the position of just one field using `Marshal.OffsetOf` and converting the data using the `BitConverter` class. [Borland.Vcl.WinUtils](#) contains helper functions to perform conversions between byte arrays and structures (see `StructureToBytes` and `BytesToStructure`).

## Special cases

There are cases where custom processing is required, such as sending a message with a pointer to an array of integers. For situations like this, the `Marshal` class provides methods to copy data directly to the unmanaged buffer, at specified offsets (so you can construct an array of a custom data type after allocating a buffer). The following example shows how to send a message where the `LParam` is a pointer to an array of `Integer`:

```

function SendArrayMessage(Handle: HWND; Msg: UINT; WParam: WPARAM;
    LParam: TIntegerDynArray): LRESULT;
var
    Buffer: IntPtr;
begin
    Buffer := Marshal.AllocHGlobal(Length(LParam) * SizeOf(Integer));
    try
        Marshal.Copy(LParam, 0, Buffer, Length(LParam));
        Result := SendMessage(Handle, Msg, WParam, Buffer);
    finally
        Marshal.FreeHGlobal(Buffer);
    end;
end;

```

```
end;  
end;
```

## Callback Functions

---

When passing a function pointer for a managed function to an unmanaged API, a reference must be maintained to the delegate or it will be garbage collected. If you pass a pointer to your managed function directly, a temporary delegate will be created, and as soon as it goes out of scope (at the end of `MyFunction` in the example below), it is subject to garbage collection. Consider the following Delphi 7 code:

```
function MyFunction: Integer;  
begin  
    ...  
    RegisterCallback(@MyCallback);  
    ...  
end;
```

In order for this to work in a managed environment, the code needs to be changed to the following:

```
const  
    MyCallbackDelegate: TFNMyCallback = @MyCallback;  
  
function MyFunction: Integer;  
begin  
    ...  
    RegisterCallback(MyCallbackDelegate);  
    ...  
end;
```

This will ensure that the callback can be called as long as `MyCallbackDelegate` is in scope.

## Data types

The same rules apply for callbacks as any other unmanaged API function.

## Special cases

Any parameters used in an asynchronous process must be declared as `IntPtr`. The marshaler will free any memory it has allocated for unmanaged types when it returns from the function call. When using an `IntPtr`, it is your responsibility to free any memory that has been allocated.

## Passing Object References

---

When working with for example the Windows API, object references are sometimes passed to the API where they are stored and later passed back to the application for processing usually associated with

a given event. This can still be accomplished in .NET, but special care needs to be taken to ensure a reference is kept to all objects (otherwise they can and will be garbage collected).

## Data types

The following table shows

Unmanaged Data Types	Managed Data Type	
	Supply Data	Receive Data
Pointer (Object reference, user data)	GCHandle	GCHandle

The GCHandle provides the primary means of passing an object references to unmanaged code, and ensuring garbage collection does not happen. A GCHandle needs to be allocated, and later freed when no longer needed. There are several types of GCHandle, GCHandleType.Normal being the most useful when an unmanaged client holds the only reference. In order pass a GCHandle to an API function once it is allocated, type cast it to IntPtr (and optionally onwards to LongInt, depending on the unmanaged declaration). The IntPtr can later be cast back to a GCHandle. Note that IsAllocated must be called before accessing the Target property, as shown below:

```
procedure MyProcedure;
var
  Ptr: IntPtr;
  Handle: GCHandle;
begin
  ...
  if Ptr <> nil then
  begin
    Handle := GCHandle(Ptr);
    if Handle.IsAllocated then
      DoSomething(Handle.Target);
  end;
  ...
end;
```

## Advanced techniques

The use of a GCHandle, although relatively easy, is fairly expensive in terms of performance. It also has the possibility of resource leaks if handles aren't freed correctly. If object references are maintained in the managed code, it is possible to pass a unique index, for example the hash code returned by the GetHashCode method, to the unmanaged API instead of an object reference. A hash table can be maintained on the managed side to facilitate retrieving an object instance from a hash value if needed.

An example of using this technique can be found in the TTreeNode class (in [Borland.Vcl.ComCtrls](#)).

## Using COM Interfaces

When using COM interfaces, a similar approach is taken as when using unmanaged API's. The interface needs to be declared, using custom attributes to describe the type interface and the GUID. Next the methods are declared; using the same approach as for unmanaged API's. The following example uses the IAutoComplete interface, defined as follows in Delphi 7:

```
IAutoComplete = interface(IUnknown)
  ['{00bb2762-6a77-11d0-a535-00c04fd7d062}']
  function Init(hwndEdit: HWND; punkACL: IUnknown;
    pwszRegKeyPath: LPCWSTR; pwszQuickComplete: LPCWSTR): HRESULT; stdcall;
  function Enable(fEnable: BOOL): HRESULT; stdcall;
end;
```

In Delphi 8 for .NET it is declared as follows:

```
[ComImport, GuidAttribute('00BB2762-6A77-11D0-A535-00C04FD7D062'),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIUnknown)]
IAutoComplete = interface
  function Init(hwndEdit: HWND; punkACL: IEnumString;
    pwszRegKeyPath: IntPtr; pwszQuickComplete: IntPtr): HRESULT;
  function Enable(fEnable: BOOL): HRESULT;
end;
```

Note the custom attributes used to describe the GUID and type of interface. It is also essential to use the ComImportAttribute class. There are some important notes when importing COM interfaces. You do not need to implement the [IUnknown/IDispatch](#) methods, and inheritance is not supported.

## Data types

The same rules as unmanaged functions apply for most data types, with the following additions:

Unmanaged Data Type	Managed Data Type	
	Supply Data	Receive Data
GUID	System.Guid	System.Guid
IUnknown	TObject	TObject
IDispatch	TObject	TObject
Interface	TObject	TObject

Variant	TObject	TObject
SafeArray (of type)	array of <type>	array of <type>
BSTR	String	String

Using the `MarshalAsAttribute` custom attribute is required for some of the above uses of `TObject`, specifying the exact unmanaged type (such as `UnmanagedType.IUnknown`, `UnmanagedType.IDispatch` or `UnmanagedType.Interface`). This is also true for certain array types. An example of explicitly specifying the unmanaged type is the `Next` method of the `IEnumString` interface. The Win32 API declares `Next` as follows:

```
HRESULT Next(
    ULONG celt,
    LPOLESTR * rgelt,
    ULONG * pceltFetched
);
```

In Delphi 8 for .NET the declaration would be:

```
function Next(celt: Longint;
    [out, MarshalAs(UnmanagedType.LPArray, ArraySubType =
UnmanagedType.LPWStr, SizeParamIndex = 0)]
    rgelt: array of string;
    out pceltFetched: Longint
): Integer;
```

## Advanced techniques

When working with safearrays, the marshal layer automatically converts (for example) an array of bytes into the corresponding safearray type. The marshal layer is very sensitive to type mismatches when converting safearrays. If the type of the safearray does not exactly match the type of the managed array, an exception is thrown. Some of the Win32 safearray API's do not set the type of the safearray correctly when the array is created, which will lead to a type mismatch in the marshal layer when used from .NET. The solutions are to either ensure that the safearray is created correctly, or to bypass the marshal layer's automatic conversion. The latter choice may be risky (but could be the only alternative if you don't have the ability to change the COM server that is providing the data). Consider the following declaration:

```
function AS_GetRecords(const ProviderName: WideString; Count: Integer;
    out RecsOut: Integer; Options: Integer; const CommandText: WideString;
    var Params: OleVariant; var OwnerData: OleVariant): OleVariant;
```

If the return value is known to always be a safearray (that doesn't describe its type correctly) wrapped in a variant, we can change the declaration to the following:

```

type
  TSafeByteArrayData = packed record
    VType: Word;
    Reserved1: Word;
    Reserved2: Word;
    Reserved3: Word;
    VArray: IntPtr; { This is a pointer to the actual SafeArray }
  end;

function AS_GetRecords(const ProviderName: WideString; Count: Integer;
  out RecsOut: Integer; Options: Integer; const CommandText: WideString;
  var Params: OleVariant; var OwnerData: OleVariant): TSafeByteArrayData;

```

Knowing that an OleVariant is a record, the TSafeByteArrayData record can be extracted from Delphi 7's TVarData (equivalent to the case where the data type is varArray). The record will provide access to the raw pointer to the safearray, from which data can be extracted. By using a structure instead of an OleVariant, the marshal layer will not try to interpret the type of data in the array. You will however be burdened with extracting the data from the actual safearray.

## Special cases

Although it is preferred to use Activator.CreateInstance when creating an instance, it is not fully compatible with [CoCreateInstanceEx](#). When working with remote servers, CreateInstance will always try to invoke the server locally, before attempting to invoke the server on the remote machine. Currently the only known work-around is to use CoCreateInstanceEx.

Since inheritance isn't supported, a descendant interface needs to declare the ancestor's methods. Below is the IAutoComplete2 interface, which extends IAutoComplete.

```

[ComImport, GuidAttribute('EAC04BC0-3791-11d2-BB95-0060977B464C'),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIUnknown)]
IAutoComplete2 = interface(IAutoComplete)
  // IAutoComplete methods
  function Init(hwndEdit: HWND; punkACL: IEnumString;
    pwszRegKeyPath: IntPtr; pwszQuickComplete: IntPtr): HRESULT;
  function Enable(fEnable: BOOL): HRESULT;
  //
  function SetOptions(dwFlag: DWORD): HRESULT;
  function GetOptions(var dwFlag: DWORD): HRESULT;
end;

```

# Building Web Applications with ASP.NET

---

ASP.NET is the programming model for building Web applications using the .NET Framework. This section provides the conceptual background for building ASP.NET applications using Delphi 8 for .NET. In addition to supporting data access components within the .NET Framework, Delphi 8 for .NET includes DB Web Controls. DB Web Controls work with .NET Framework providers and Borland Data Providers for .NET (BDP.NET) to accelerate Web application development.



# ASP.NET Overview

---

ASP.NET is the .NET programming environment for building Internet applications with an HTML front end. The ASP.NET architecture is designed for seamless integration with a number of .NET programming models, including its own Web Forms for the front-end interface, Web Services as an optional approach to xml-based messaging, and ADO.NET for optional, back-end data access. Use ASP.NET to build forms-based applications in HTML that run on the Web.

Borland provides tools to simplify rapid ASP.NET development. If you are familiar with rapid application development (RAD) and object oriented programming (OOP) using properties, methods, and events, you will find the ASP.NET model for building rich Web applications familiar.

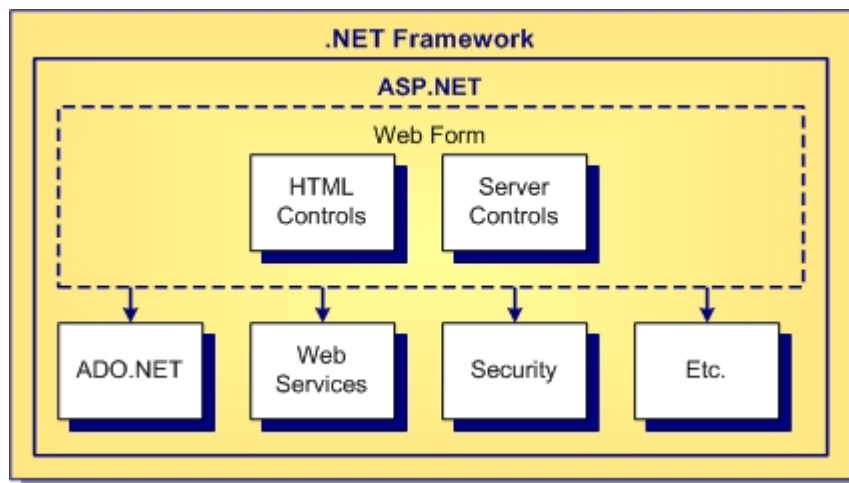
The following topic provides the conceptual background 1) to understand the major components of the ASP.NET architecture and 2) to understand how ASP.NET integrates with other programming models in the .NET framework.

This section introduces:

- ASP.NET architecture.
- Web Forms components.
- Web Forms data access.
- Web Services.
- ASP.NET namespace.
- ASP.NET application deployment.

## ASP.NET Architecture

---



The major components of the ASP.NET architecture are the Web Forms page, ASP.NET Server Controls, code-behind logic, and compiled DLL files. Web Form pages contain HTML elements, text, and server controls. Code-behind files contain application logic for the Forms page. Compiled DLL files render dynamic HTML on the web server.

## **Web Forms Components**

---

Web Forms define the user interface for your Web application using HTML for the visual presentation with code-behind logic stored in separate files for compilation. Web Forms consist of pages, controls, and code-behind files.

### **Web Forms Page**

The Web Forms page contains HTML and ASP.NET server controls. When a user first accesses the Forms page over the web, ASP.NET generates a compiled DLL file that then dynamically generates HTML output for browsing.

### **HTML Elements and Server Controls**

HTML elements in ASP.NET closely match many common HTML elements found on web pages. Besides common HTML elements, Web Forms pages often contain ASP.NET server controls. ASP.NET server controls fall under a number of categories: such as Web, validation, and user controls. Web server controls match to form-type controls, plus special-purpose controls like a calendar or a data grid. Validation controls allow you to test user input. User controls provide a mechanism for creating reusable components like menus and toolbars. Once created, you can embed user controls in other Web Forms pages.

### **Code-behind File**

The code-behind file contains the application logic that interacts with the visual components and ASP.NET server controls in your Web Forms page.

## **Web Forms Data Access**

---

Within the .NET Framework, Web Forms access data via ADO.NET. You can connect an ASP.NET application to an ADO.NET data source using data components included in the .NET Framework or BDP.NET components included in the product. BDP.NET components connect to a number of industry standard databases. In addition, DB Web Controls simplify database development tasks whether using BDP.NET or .NET Framework data access components. DB Web Controls provide you with advanced functionality, including data-aware grid, navigator, calendar, combobox, and other popular components.

## **Web Services**

---

Web Services provide application components to any number of distributed systems using XML-based messaging. A Web Service can be as simple as an XML message updating values in a remote

application (perhaps a stock quote in a web client) to an integral part of a sophisticated ASP.NET or ADO.NET application. Web Services and ASP.NET share common .NET infrastructure that allows for seamless integration.

## **ASP.NET Namespace**

---

See the Microsoft .NET Framework SDK for information pertaining to the System.Web namespaces and ASP.NET.

## **ASP.NET Application Deployment**

---

After creating an ASP.NET project with Delphi 8 for .NET, mount the aspx files and the bin directory with assemblies on a Web server with the appropriate permissions applied to a virtual directory. The project (\*.bdsproj) and code-behind files (\*.pas) are not necessary for HTTP. The logic within the code-behind file is compiled in the assemblies located within the \bin directory. The project and code-behind files are needed only for maintenance within Delphi 8 for .NET. Typically, ASP.NET developers create and test Web projects locally, then mount the required subset of ASP.NET files on the HTTP server.

When deploying a Delphi 8 for .NET ASP.NET application, you need to deploy the Delphi 8 for .NET compiler as well, with the application. This allows the Delphi 8 for .NET scripts in your ASP.NET application to be compiled as needed. For more information, refer to the DEPLOY document included in the Delphi 8 for .NET installation.

# Borland DB Web Controls

---

Borland DB Web Controls simplify database development tasks in conjunction with BDP.NET and .NET Framework data access components. DB Web Controls are data-aware controls that provide advanced functionality, including data-aware grid, navigator, calendar, combobox, and other popular components.

This section introduces:

- DB Web Controls architecture.
- Data-aware components advantages.
- Supported data access components.
- DB Web Controls namespace.
- ASP.NET application deployment with DB Web Controls.

## DB Web Controls Architecture

---

Like other .NET database technologies, DB Web Controls exist within a multi-tier architecture. DB Web Controls are common GUI controls for ASP.NET applications that reference DBWebDataSource for their data. The DBWebDataSource, in turn, references .NET Framework and BDP.NET data access components (for example, DataSet, DataView, DataTable, and BdpDataAdapter). Though not a requirement, in most cases the Dataset connects to a DataProvider, which uses a Connection to the data source. For more information about data access components, see the chapter on Borland Data Providers for .NET.

## Data-aware Components Advantages

---

The multi-tier architecture simplifies development in ASP.NET. Once a DB Web Control connects to a DBWebDataSource, a databind call is not needed. The components are now data-aware. Fully leveraging the features of the underlying ASP.NET controls, DB Web Controls provide a number of advantages:

1. Do not require a call to WebControl.databind(). Normally, each ASP.NET control on the web form requires this call from the Page\_load method or the control will not appear on the form.
2. Provide a design-time view of the data.
3. Automatically post changes back to the dataset. Typically, ASP.NET controls require code to post back changes.
4. Maintain current row position.
5. Change and row state are managed automatically using sessions. This means that clients from different machines will operate independently.
6. The DBWebDataSource maintains an ordered list of changes, so that the user can undo changes in the order in which they were made.
7. Provide a data-aware navigator control.

8. In several places where you would otherwise need to write code with DataGrid (for example, paging using numbers, paging using the previous and next icons, Edit, and Delete columns), no code is required using DBWebDataGrid.

## **Supported Data Access Components**

---

DB Web Controls are compatible with .NET Framework and BDP.NET data access components.

## **DB Web ControlsNamespace**

---

The namespace for DB Web Controls is Borland.Data.Web.

## **ASP.NET Application Deployment with DB Web Controls**

---

After creating an ASP.NET project with DB Web Controls, deploy your ASP.NET application as usual. No special considerations are required.

# Building Web Services with ASP.NET

---

Web Services is a programmable entity that provides a particular element of functionality, such as application logic, and is accessible to any number of potentially disparate systems through the use of Internet standards, such as XML and HTTP. Applications built with ASP.NET Web Services can be either stand-alone applications or sub-components of a larger web application and can provide application components to any number of distributed systems using XML-based messaging. Delphi 8 for .NET provides a number of methods that can help you build, deploy, and use applications with ASP.NET Web Services. For more general information about Web Services, refer to Microsoft's .NET SDK Documentation.

# ASP.NET Web Services Overview

---

Web Services is an Internet-based integration methodology that enables applications, independent of any platform or language, to connect and exchange information. Web Services are tightly integrated with the .NET Framework's ASP.NET model. Unlike traditional native Windows applications, ASP.NET Web Services applications contain objects and methods that are exposed over the Web using simple messaging protocol stacks. Any client can invoke a Web Services application over HTTP using a WebMethod. Like any method that can be accessed by way of a simple Windows Form application, a WebMethod provides some defined functionality, but unlike other types of methods, the WebMethod is accessed by way of a web browser. For more general information about Web Services, refer to Microsoft's .NET Framework SDK Documentation.

Borland provides tools to develop and access ASP.NET Web Services using a variety of techniques. As modular objects, Web Services provide reuse without additional coding.

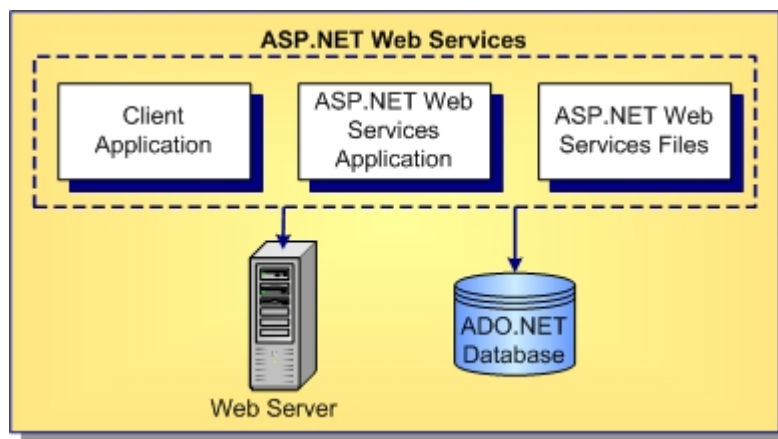
The following topics provide a brief introduction to the architecture of ASP.NET Web Services, the basic fundamentals of Web Services communication, and to the files created when you develop ASP.NET Web Services.

This section introduces:

- ASP.NET Web Services Architecture.
- Web Services Prerequisites.
- Web Services Scenarios.
- ASP.NET Web Services Files.

## Web Service Architecture

---



The major components of the ASP.NET Web Services architecture include a client application, an ASP.NET Web Services application, and several files such as code files in the development language, .asmx files, and compiled .dll files. You need a web server to house both ASP.NET Web Services application and the client. Optionally, you might include a database server for storage and access of ASP.NET Web Services data.

## Web Service Prerequisites

---

Before you begin developing a Web Services application, become familiar with the following concepts:

- **XML (Extensible Markup Language).** XML is a user-defined, human-readable structural description of data. Any data, dataset, or document that you intend to send to or receive from a web service is formatted in XML.
- **SOAP (Simple Object Access Protocol).** SOAP is the standard messaging protocol that is used for communication between web services and their clients. SOAP uses XML to format its messages, and contains the parameters or return values needed by servers and clients.
- **WSDL (Web Services Description Language).** WSDL is the language that describes a web service. A web service can be defined in any number of implementation languages. As a single-purpose utility, each web service must publish a description of its interface, which allows clients to interact with it. The WSDL document, at a minimum, describes the required parameters a client must provide and the result a client can expect to receive. The result description typically consists of the return data type.
- **UDDI (Universal Description, Discovery, and Integration).** UDDI is an industry initiative that provides a standard repository where businesses can publish web services for use by other companies. The UDDI repository contains links to and descriptions of a variety of web services. You can use the UDDI browser in the IDE to locate web services, download WSDL documents, and access additional information about web services and the companies that provide them.

## Web Service Scenarios

---

Current Web Services provide simple information sources that you can easily incorporate into applications, such as stock quotes, weather forecasts, and sports scores. As the demand for access to business logic over the web increases, companies are finding ways of providing their customers with a class of applications to analyze and aggregate information. If a customer now keeps a Microsoft Excel spreadsheet in which they manually summarize their financial information, such as stock portfolio, 401 (k), bank account, and loan information, in the future, a Web Service will consolidate and continuously update this information for display by Excel or on a personal web site or digital assistant. Although much of this information is available through the web today, a Web Service will simplify access and consolidation and will ensure greater reliability.

You can use Web Services for solutions in the following areas:



- **Enterprise Application Integration (EAI).** A web service could allow multiple business partners to exchange inventory, order, or financial data, for example, without specifically knowing the precise data layout in which each partner's data is stored. For instance, many CRM or other front-end applications store customer data in a format that is not entirely compatible with the way a back-end ERP system stores its financial or inventory information. Yet, a sales organization may wish to use its CRM solution to process real-time orders with up-to-date inventory information from the ERP system. A web service could be a solution to managing the transformation of CRM requests to ERP storage and from ERP responses to CRM confirmations.
- **Business-to-business (B2B) integration.** Similar to the EAI solution, a B2B solution could take advantage of a web service capability to provide cached data for large orders. B2B transactions, unlike business-to-consumer (B2C) transactions, often consist of high-volume transactions that would be prohibitive to execute at the granular level of a B2C transaction. For instance, a consumer might order one box of pencils from an online stationery store, but a business might order a thousand boxes monthly, with multiple shipping addresses. The scale and complexity of a B2B transaction requires the intervention of a web service to help simplify and process the transaction quickly and with consistency.
- **Business-to-consumer (B2C) integration.** B2C web services typically manage web-based transactions. For example, a web service that allows you to look up postal codes eliminates the need for any given individual to create a new program to perform this task every time they want to include the service on a web site. Some commerce sites might use web services to help manage currency conversion when taking international sales orders.
- **Mobile (Smart client applications).** Because the small footprint of a mobile client requires that memory usage be reserved for only the most important system functions, and because mobile clients are, by definition, linked to the Internet by way of their wireless communication protocols, web services play a vital role in providing lightweight but powerful applications to mobile devices. Web services allow mobile device users to perform a variety of tasks which require little more than data input at the device and data display of the results. All processing can occur on a remote web service, thus decreasing bandwidth requirements on the mobile device itself.
- **Distributed/Peer-to-Peer.** For certain types of distributed and peer-to-peer applications, web services play an important role. If using distributed computing over an uncontrolled network (such as the Internet) rather than over a LAN or corporate network, you might use web services, which do not require state maintenance, thus offering potentially improved performance, particularly where a request-response behavior is not unequivocally required. For applications that require strict request-response behavior and high security, you should consider using an older, more controlled model, such as COM, CORBA, or .NET remoting.

## ASP.NET Web Services Files

---

Certain files are automatically generated when you create applications with ASP.NET Web Services. These files enable the ASP.NET Web Services to render their services through a web server. The following table lists the name of the files and their descriptions.

Name of files	Description
.asmx	When you create an ASP.NET Web Services application, a text file is automatically generated with the .asmx extension. Additionally, the required Web Services directive is placed at the top of this file to correlate between the URL address of the Web Services and its implementation. Within the .asmx file, you add Web Services logic to the methods visible by the client application. The .asmx file acts as the base URL for clients calling the XML Web Service. This file is compiled into an assembly, along with other files, for deployment.
code-behind	When you create an ASP.NET Web Service application, a code-behind file is generated with a language-specific extension. You add your Web Services logic to the public method to process Web Services requests and responses.
compiled DLL filesWeb Services DLL files provide dynamic services on the web server.	
.wsdl	This file is generated when you click the <i>Add Web Reference</i> feature to add the Web Services to your client application. It describes the Web Services interface available to the client.
.map	This file enables the discovery of a Web Service that is exposed on a given server. It also contains links to other resources that describe the Web Service.

# Web Services Protocol Stack

---

Understanding the Web Services infrastructure requires that you have some exposure to XML, SOAP, WSDL, and UDDI. As a developer of XML Web Services, you are not creating anything from the ground up because the infrastructure already exists. Instead, you leverage the existing technology by using standard Web protocols such as XML and HTTP.

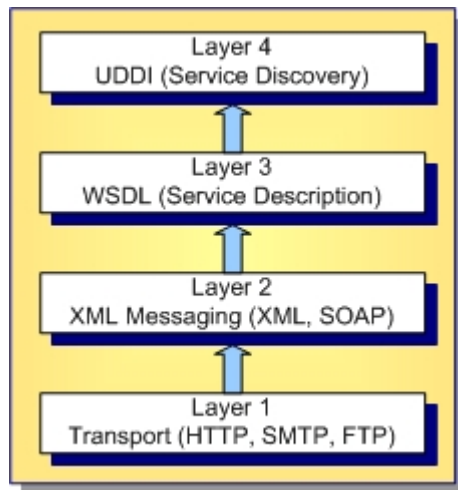
Borland provides an easy way to create, deploy, and use Web Services without the worries of back-end processing so you can focus more on designing your services.

The following topics provide the conceptual background to understand how the protocol stack contributes to Web Services functionality.

- How Web Services access and expose their services via the Web.
- How XML passes information through standard SOAP and HTTP.
- How a client can identify a Web Service offering.
- How Web Services are discovered and accessed.

## Web Services Protocol Stack

---



The major components of a Web Service Protocol Stack include:

- A layer for transporting messages between applications.
- A layer to encode messages in XML that can be understood by both client and server.
- A layer to describe the service provided.
- A common registry that centralizes services.

## Transport Layer

---

The Transport layer is the first component in the stack and is responsible for moving XML messages between applications. The Transport protocol most commonly used is the standard HTTP protocol. Other commonly used Web protocols are SMTP and FTP.

## XML Messaging

---

The messaging layer in the protocol stack is based on an XML model. XML (Extensible Markup Language) is widely used in Web Services applications and is the foundation for all Web Services. XML is just one of the standards enabling Web Services to map between technology domains. You will find many resources on the Web that describe XML messaging. For more information, refer to the World Wide Web Consortium site on Messaging listed in the link list below.

The XML Messaging specification is a broadly-defined umbrella under which a number of more specific protocols are defined. One of the more popular standards is known as the Simple Object Access Protocol (SOAP) and is one of the most significant standards in communicating Web Services over the network. Just as XML provides a means for communicating over the Web using an XML document that both requests and responds to information between two disparate systems, SOAP allows the sender and the receiver of XML documents to support a common data transfer protocol for effective networked communication. You will find many resources on the Web that describe SOAP. For more information, refer to the World Wide Web Consortium site for SOAP. See the link list below.

## WSDL Layer

---

This layer represents a way of specifying a public interface for a Web Service. It contains information on available functions, on data types for XML messaging, binding information about the transport protocol, and the location of the specific Web Service.

Any client application that wants to know about a service, what data it expects to receive, whether or not it delivers any results, and the supported transport, uses the Web Services Description Language (WSDL) to find that information. When you create a Web Service, it must be described and advertised to its potential customers before it can be used. WSDL provides a common format for describing and publishing that Web service information. Typically, WSDL is used with SOAP, and the WSDL specification includes a SOAP binding.

Use Borland's *Add Web Reference* feature to obtain a WSDL document for your Web Service. The WSDL document, or proxy file, is copied to the client and is used to call the server. This proxy file is called *References.\** and has an extension that reflects its particular language type. For more information about WSDL, refer to the World Wide Web Consortium site.

## Service Discovery

---

Service Discovery is a layer that represents a way to publish and find Web Services over the Web. You can think of this layer as the White and Yellow Pages of your phonebook. The White pages of Web Services provides general information about a specific company, for instance, their business name, description, and address. The Yellow Pages includes the classification of data for the services offered, for instance, industry type and products.

The protocol you use to publish your Web Services is known as Universal Description, Discovery, and Integration (UDDI).

With Delphi 8 for .NET, your data automatically gets published to the registry, or a distributed directory for business and Web Services. The UDDI Business Registry allows anyone to search existing UDDI data and enables you to register your company and its services.

# ASP.NET Web Services Support

---

ASP.NET Web Services support VCL.NET Forms, .NET Windows Forms, and ASP.NET Web Forms, which can be used to create client applications that access Web Services applications. An *Add Web Reference* feature is provided to add the desired ASP.NET Web Services application to the client application. Also available by way of the *Add Web Reference* feature, a *UDDI Browser* helps you locate Web Services applications you might want to use.

Delphi 8 for .NET provides simple tools to develop and deploy your ASP.NET Web Services applications. Additionally, Delphi 8 for .NET helps you import WSDL documents that describe particular Web Services applications and expose their functionality to the client application.

You can use the sample WebMethod provided by Delphi 8 for .NET, which lets you create and access an ASP.NET Web Services application.

This section includes:

- ASP.NET Web Services Client Support.
- ASP.NET Web Services Server Support.
- ASP.NET Web Services Namespaces.

## ASP.NET Web Services Client Support

---

You can create a Web Services application that is nothing more than a provider, or a server application. This application resides on a web server and can be accessed by any client that understands the application architecture. If you want to consume a Web Services application yourself, you need to create a client application. Delphi 8 for .NET provides different tools you can use to build client applications:

- Windows Forms
- Web Forms
- Web References

## Windows Forms vs. ASP.NET Web Forms

To determine the best type of form to use for your client application, consider the type of service you want to access. In most cases, the service you choose will dictate which type of application you should create.

If you need to provide a rich application that can process complex content on a client workstation, or that can use a Web Service application as a supporting piece for a rich client application over a secure network connection, you might consider building a Windows Forms application. If you need to provide a thin-client application that performs simple data manipulation or satisfies a single-purpose requirement, consider using ASP.NET Web Forms. Web Forms are platform-independent interfaces that display in a web browser and invoke Web Services applications over a simple protocol like HTTP.

You can also create an ASP.NET Web Services application as a console application which can be accessed through either a console window, or by another Web Service application, even one without a client.

## Add Web Reference

You can add a Web Reference to your client application to access Web Services applications. A Web Reference refers to either a WSDL document or an XML schema, which is imported into your client application. The WSDL document or XML schema describes a Web Service application. When you import one of these documents, Delphi 8 for .NET generates the interfaces and class definitions needed for calling that Web Service application. Right-click the *WebService* node in the *Project Manager* and select *Add Web Reference*. A *UDDI Browser* appears. You must navigate within the browser and locate your Web Service's WSDL document in order to add the Web Services to your client application.

## ASP.NET Web Services Server Support

---

When you build an ASP.NET Web Services application, you are providing a way to programmatically access the application logic of the Web Service. You define the service you want to expose, how the service is to be used, and the infrastructure that receives and processes requests and responses.

Delphi 8 for .NET provides a dialog that lets you specify the name and location of the ASP.NET Web Services application, and automatically creates the mandatory files that are necessary for deployment. When you specify the application settings, Delphi 8 for .NET generates the .asmx file that acts as a base URL for clients calling the ASP.NET Web Services application.

## ASP.NET Web Services Namespaces

---

For more information on System.Web.Services namespaces, refer to the Microsoft .NET Framework SDK.

# Building Windows Applications with Windows Forms

---

Windows Forms provide a traditional approach to developing user interfaces, client/server applications, forms, controls, and application logic. Windows Forms fully leverages the .NET Framework. This section provides an overview of Windows Forms using Delphi 8 for .NET and common steps to building a simple Windows project.



# Windows Forms Overview

---

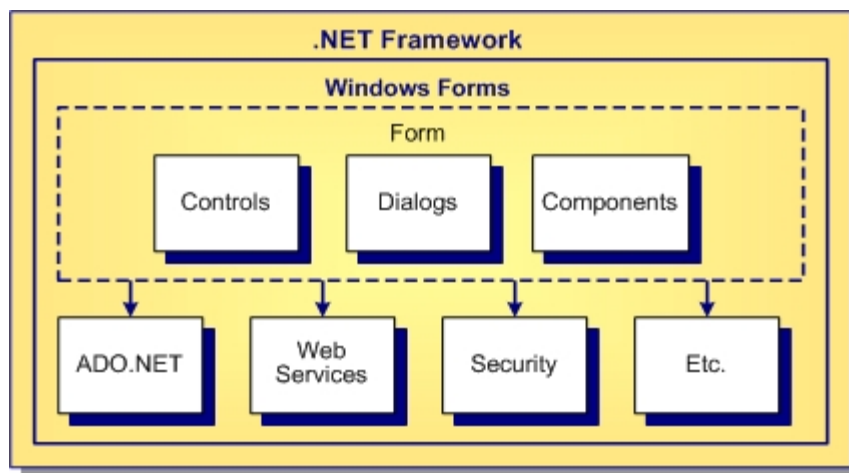
Windows Forms is the .NET programming environment for building native Windows applications in a managed environment. Building Windows clients with .NET allows applications to use features unavailable to browser clients while leveraging the .NET Framework for general infrastructure. Windows Forms combines features of both traditional and Internet-centric development, presenting a programming model that takes advantage of a unified .NET Framework (for instance, for security and dynamic application updates) and the richness of GUI Windows clients.

This section introduces:

- Windows Forms architecture.
- Windows Forms components.
- Windows Forms data access.
- Windows Forms namespace.

## Windows Forms Architecture

---



Windows Forms share common .NET Framework with other programming models, like ASP.NET and ADO.NET.

## Windows Forms

---

Delphi 8 for .NET provides an IDE for creating a GUI in a RAD environment. Developers drag and drop controls, dialogs, and components onto the form designer, set properties in *Object Inspector*, and code the logic to respond to events.

The Windows Form object in Delphi 8 for .NET is called `TWinForm` and is a descendant of `TForm`. This differs from the VCL.NET-based form that you can build using the VCL.NET designer. In effect, the `TWinForm` object is a .NET Windows Form that uses Object Pascal as its code-behind.

## Windows Forms Components

---

Delphi 8 for .NET's *Tool Palette* for Windows Forms provides drag and drop components, controls, and dialogs for designing a GUI. Components are classes that represent reusable objects with drag and drop functionality in a RAD IDE. Controls are a type of component with user interface functionality. (All controls are components, but not all components are controls.) Add components to the designer surface. Typically, you design user interfaces by positioning and sizing components and controls on a form. Examples of common controls and components include buttons and menus. To facilitate the construction of menus, Delphi 8 for .NET provides a menu designer for `MainMenu` and `ContextMenu` components. Dialog boxes are a type of form, which in turn can contain controls. Dialogs provide for various types of user interaction.

## Windows Forms Data Access

---

Within the .NET Framework, Windows Forms access data via ADO.NET. You can connect a Windows application to an ADO.NET data source using data components included in the .NET Framework and BDP.NET. BDP.NET components connect to a number of industry standard databases. For more information, see the ADO.NET section.

## Windows Forms Namespace

---

Common Windows Forms classes like `Form` and `Menu` are contained within the `System.Windows.Forms` namespace. The namespace also contains controls like `Button`, `CheckBox`, and `Label`. Use Delphi 8 for .NET's *Object Inspector* to set properties, methods, and events within Windows Forms classes.

## Windows Forms Application Deployment

---

For the common language runtime, deploying Windows Forms applications requires installation of the .NET Framework on the target computer. If the Windows Forms application is simple, consisting of a single executable, the .exe file may reside unregistered in the appropriate program directory. If the Windows Forms application includes a shared assembly, the assembly must be installed to the Global Assembly Cache using tools in the .NET Framework. For more information, see the .NET Framework SDK help.

# Building Database Applications with ADO.NET

---

ADO.NET presents a coherent programming model for exposing data access within the .NET Framework. In addition to supporting MS SQL, Oracle, and OLE DB connection components within the .NET Framework, Delphi 8 for .NET includes Borland Data Providers for .NET (BDP.NET). BDP.NET supports access to MS SQL, Oracle, DB2, and Interbase. BDP.NET component designers ease the generation and configuration of BDP.NET components. This section overviews how to use Delphi 8 for .NET with the ADO.NET architecture and how to build a simple ADO.NET project.

# ADO.NET Overview

---

ADO.NET is the .NET programming environment for building database applications based on native database formats or XML data. ADO.NET is designed as a back-end data store for all .NET programming models, including Web Forms, Web Services, and Windows Forms. Use ADO.NET to manage data in the .NET Framework.

Borland provides tools to simplify rapid ADO.NET development using Borland Data Providers for .NET (BDP.NET). If you are familiar with rapid application development (RAD) and object oriented programming (OOP) using properties, methods, and events, you will find the ADO.NET model for building applications familiar. If you are a traditional database programmer, ADO.NET provides familiar concepts, such as tables, rows, and columns with relational navigation. XML developers will appreciate navigating the same data with nodes, parents, siblings, and children.

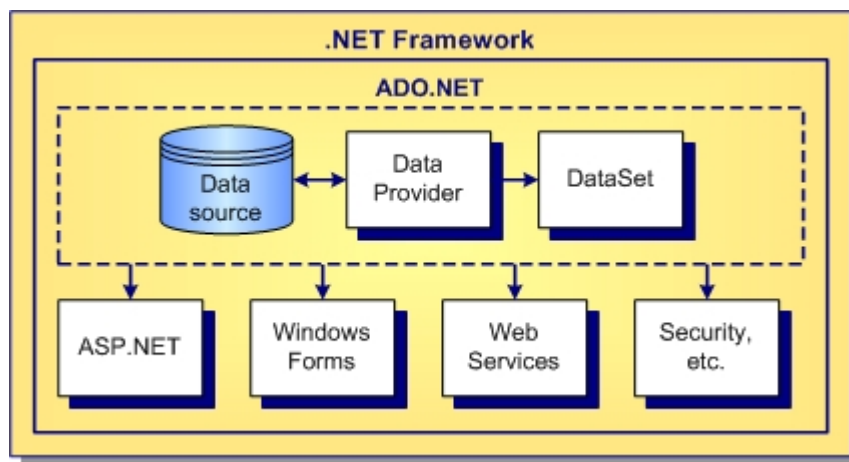
The following topic provides the conceptual background to understand 1) the major components of the ADO.NET architecture, 2) how ADO.NET integrates with other programming models in the .NET Framework, and 3) key Delphi 8 for .NET functionality.

This section introduces:

- ADO.NET architecture.
- ADO.NET user interfaces.
- TADONETConnector object.
- BDP.NET namespace.
- ADO.NET application deployment.

## ADO.NET Architecture

---



The two major components of the ADO.NET architecture are the Data Provider and the DataSet. The data source represents the physical database or XML file, the Data Provider makes connections and passes commands, and the DataSet represents one or more data sources in memory. For more information about the general ADO.NET model, see the .NET Framework Developer's Guide.

## Data Source

The data source is the physical database, either local or remote, or an XML file. In traditional database programming, the developer typically works with the data source directly, often requiring complex, proprietary interfaces. With ADO.NET, the database developer works with a set of components to access the data source, to expose data, and to pass commands.

## Data Providers

Data Provider components connect to the physical databases or XML files, hiding implementation details. Providers can connect to one or more data sources, pass commands, and expose data to the DataSet.

The .NET Framework includes providers for MS SQL, OLE DB, and Oracle. In addition to supporting the .NET providers, this product includes Borland Data Providers for .NET. BDP.NET connects to a number of industry standard databases, providing a consistent programming environment. For more information, see the Borland Data Providers for .NET topic.

Delphi 8 for .NET includes support for the .NET data providers by way of the TADONETConnector object. This object provides access to .NET DataSets either directly or through the Borland Data Providers for .NET. BDP.NET connects to a number of industry standard databases, providing a consistent programming environment. For more information, see the Borland Data Providers for .NET topic and the reference information for TADONETConnector.

TADONETConnector is the base class for Delphi 8 for .NET datasets that access their data using ADO.NET. TADONETConnector descendants include TCustomADONETConnector. TADONETConnector is a descendent of TDataSet.

## DataSet

The DataSet object represents in-memory tables and relations from one or more data sources. The DataSet provides a temporary work area or virtual scratch pad for manipulating data. ADO.NET applications manipulate tables in memory, not within the physical database. The DataSet provides additional flexibility over direct connections to physical databases. Much like a typical cursor object supported by many database systems, the DataSet can contain multiple DataTables, which are representations of tables or views from any number of data sources. The DataSet works in an asynchronous, non-connected mode, passing update commands by way of the Provider to the data source at a later time.

Delphi 8 for .NET provides two kinds of DataSets for your use: standard DataSets and typed DataSets. A standard DataSet is the default DataSet that you get when you define a DataSet object implicitly. This

type of DataSet is constructed based on the layout of the columns in your data source, as they are returned at runtime based on your Select statement.

Typed DataSets provide you more control over the layout of the data you retrieve from a data source. A typed DataSet derives from a DataSet class. The typed DataSet lets you access tables and columns by name rather than by way of collection methods. The typed DataSet feature provides better readability, improved code completion capabilities, and data type enforcement unavailable with standard DataSets. The compiler checks for type mismatches of typed DataSet elements at compile-time rather than runtime. When you create a typed dataset, you will see that some new objects are created for you and are accessible by way of the *Project Manager*. You will notice two files named after your dataset. One file is an XML .xsd file and the other is a code file, in the language you are using. All of the data about your dataset, including the table and column data from the database connection, is stored in the .xsd file. The program code file is created based on the XML in the .xsd file. If you want to change the structure of the typed dataset, you can change items in the .xsd file. When you recompile, the program code file is regenerated based on the modified XML.

For more information about DataSets, see the Microsoft .NET Framework SDK.

## **ADO.NET User Interfaces**

---

ADO.NET provides data access for the various programming models in .NET.

### **Web Forms**

Web Forms in ASP.NET provide a convenient interface for accessing databases over the web. ASP.NET uses ADO.NET to handle data access functions.

.NET and BDP.NET connection components ease integration between Web Forms and ADO.NET. DB Web Controls support both ADO.NET and BDP.NET components, accelerating web application development.

### **Windows Forms**

As an alternative to Web Forms, traditional, native-OS clients can function as a front end to ADO.NET databases.

In Delphi 8 for .NET you can provide two types of Windows Forms: a TWinForm object, which is a descendant of TForm and acts as the native .NET Windows Form, and a VCL.NET form.

## **BDP.NET Namespace**

---

BDP.NET classes are found under the Borland.Data namespace.

### ***BDP.NET Namespace***

Namespace	Description
Borland.Data.Common	Contains objects common to all Borland Data Providers, including Error and Exceptions classes, data type enumerations, provider options, and Interfaces for building your own Command, Connection, and Cursor classes.
Borland.Data.Provider	Contains key BDP.NET classes like BdpCommand, BdpConnection, BdpDataAdapter, and others that provide the means to interact with external data sources, such as Oracle, DB2, Interbase, and MS SQL Server databases.
Borland.Data.Schema	Contains Interfaces for building your own database schema manipulation classes, as well as a number of types and enumerators that define metadata.

## ADO.NET Application Deployment

When building database applications using Delphi 8 for .NET, copy the necessary files (assemblies and DLLs) for deployment to a specified location. The following table lists the name of the assemblies and DLLs and the location of where each assembly should reside.

For specific database runtime assemblies, copy them to the following location:

Managed Assemblies	Data Provider	Location
Borland.Data.Common.dll	All	GAC
Borland.Data.Provider.dll	All	GAC
Borland.Data.DB2.dll	DB2	GAC
Borland.Data.Interbase.dll	Interbase	GAC
Borland.Data.Mssql.dll	MS SQL/MSDE	GAC
Borland.Data.Oracle.dll	Oracle	GAC

For database DLLs, copy them to the following location:

DLLs	Data Provider	Location
bdpint.dll	Interbase	search path
bdpdb2.dll	DB2	search path
bdpmss.dll	MS SQL/MSDE	search path
bdpora.dll	Oracle	search path

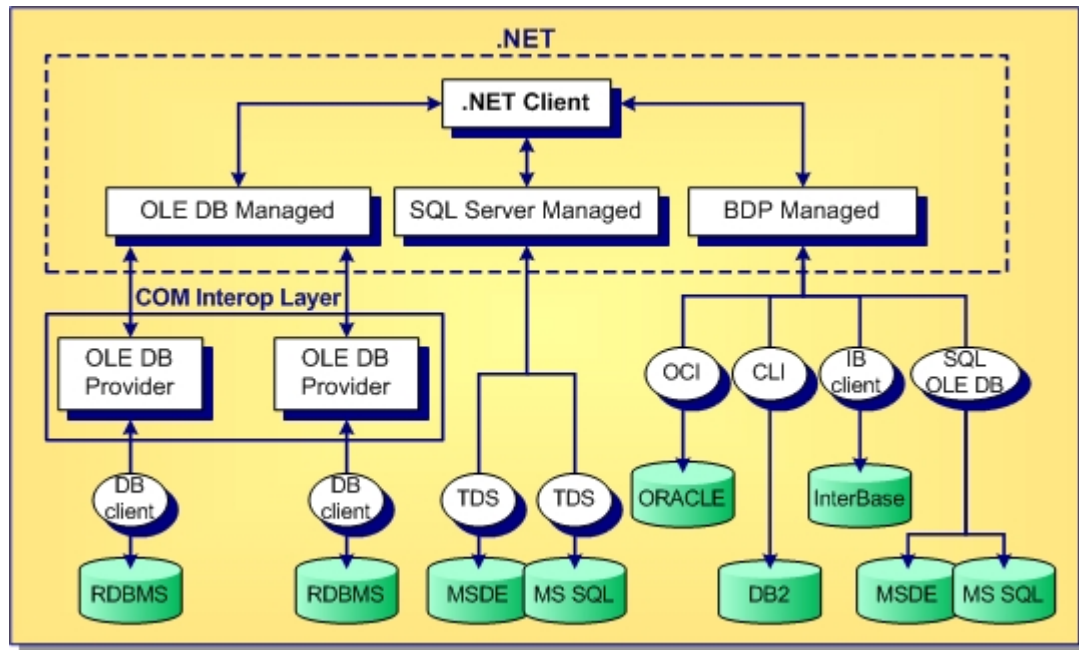
# Borland Data Providers for Microsoft .NET

In addition to supporting the providers included in the .NET Framework, Delphi 8 for .NET includes Borland Data Providers for Microsoft .NET (BDP.NET). BDP.NET, an implementation of the .NET Provider, connects to a number of popular databases. This topic covers:

- Data Provider architecture.
- BDP.NET advantages.
- BDP.NET and ADO.NET components.
- BDP.NET data types.
- BDP.NET interfaces.

## Data Provider Architecture

Delphi 8 for .NET supports the .NET Framework providers, plus providers included in BDP.NET.



BDP.NET provides a high performance architecture for accessing data sources without a COM Interop layer.



## BDP.NET Advantages

---

BDP.NET provides a number of advantages:

- A unified programming model applicable to multiple database platforms.
- A high performance data-access architecture.
- The open architecture supports additional databases easily.
- Portable code to write once and connect to any supported databases.
- Consistent data type mapping across databases where applicable.
- Logical data types mapped to .NET native types.
- Unlike OLE DB, no need for a COM Interop layer.

Delphi 8 for .NET extends .NET support to additional database platforms, providing a consistent connection architecture and data type mapping.

## BDP.NET and ADO.NET Components

---

All databases and connection components expose data in ADO.NET's DataSet component. The DataSet is the common denominator for all databases, presenting a unified programming model with tables and relational navigation. BDP.NET providers encapsulate implementation details for each database, while still providing high performance data-access.

## BDP.NET Data Types

---

The .NET Framework includes a wide range of logical data types. BDP.NET inherits logical data types, providing built-in mappings to supported databases. For more information, see the BDP.NET Data Types topic.

## BDP.NET Interfaces

---

You can extend BDP.NET to support other DBMS by implementing a subset of the .NET Provider interface. BDP.NET generalizes much of the functionality required to implement data providers. While the .NET Framework gives you the capabilities to create individual data providers for each data source, Borland has simplified the task by offering a generalized set of capabilities. Instead of building separate providers, along with corresponding DataAdapters, DataReaders, Connection objects, and other required objects, you can implement a set of BDP.NET interfaces to build your own data source plug-ins to the Borland Data Provider.

Building plug-ins is a much easier task than building a completely new data provider. You build an assembly that contains the namespace for your provider, as well as classes that encapsulate provider-specific functionality. Much of the functionality you need to connect to, execute commands against, and retrieve data from your data source has already been defined in the Borland Data Provider interfaces.

# BDP.NET Data Types

---

BDP.NET data types map to .NET logical types. Dependant upon the database, BDP.NET data types map to native data types. Where applicable, BDP.NET provides:

- Consistent data type mapping across databases.
- Logical data types mapped to .NET native types.

## BDP.NET and .NET Framework

---

The DataSet class within ADO.NET uses .NET Framework data types. BDP.NET data types logically map .NET data types for supported databases. During design time, you can use BDP.NET logical types, which will map to the appropriate native type.

## Data Types

---

The .NET Framework includes a wide range of logical data types. BDP.NET inherits logical data types, providing built-in mappings to supported databases. BDP.NET supports logical data type mappings for DB2, Interbase, MS SQL, MSDE, and Oracle.

## DB2

---

BDP.NET supports the following DB2 type mappings.

### ***BDP.NET logical data types for DB2***

DB2 Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
BIGINT	Int64	NA	Int64
INTEGER	Int32	NA	Int32
DOUBLE	Double	NA	Double
FLOAT	Float	NA	Single
REAL	Float	NA	Single
DATE	Date	NA	DateTime
TIME	Time	NA	DateTime
TIMESTAMP	Datetime	NA	DateTime
NUMERIC	Decimal	NA	Decimal

DECIMAL	Decimal	NA	Decimal
BLOB	Blob	stBinary	Byte[]
CLOB	Blob	stMemo	Char[]

## Interbase

BDP.NET supports the following Interbase type mappings.

### ***BDP.NET logical data types for Interbase***

Interbase Type	.Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
INTEGER	Int32	NA	Int32
FLOAT	Float	NA	Single
DOUBLE	Double	NA	Double
BLOB Sub_Type 0Blob		stBinary	Byte[]
BLOB Sub_Type 1Blob		stMemo	Char[]
TIMESTAMP	Datetime	NA	DateTime

## MS SQL and MSDE

BDP.NET supports the following MS SQL and MSDE type mappings.

### ***BDP.NET logical data types for Oracle***

MSSQL Type	Bdp Type	BdpSubType	System.Type
BIGINT	Int64	NA	Int64
INT	Int32	NA	Int32
SMALLINT	Int16	NA	Int16
TINYINT	Int16	NA	Int16
BIT	Boolean	NA	Boolean
DECIMAL	Decimal	NA	Decimal
NUMERIC	Decimal	NA	Decimal
MONEY	Decimal	NA	Decimal

SMALLMONEY	Decimal	NA	Decimal
FLOAT	Double	NA	Double
REAL	Float	NA	Single
DATETIME	DateTime	NA	DateTime
SMALLDATETIME	DateTime	NA	DateTime
CHAR	String	stFixed	String
VARCHAR	String	NA	String
TEXT	Blob	stMemo	Char[]
BINARY	VarBytes	NA	Byte[]
VARBINARY	VarBytes	NA	Byte[]
IMAGE	Blob	stBinary	Byte[]
TIMESTAMP	VarBytes	NA	Byte[]
UNIQUEIDENTIFIER	Guid	NA	Guid

## Oracle

BDP.NET supports the following Oracle type mappings.

### ***BDP.NET logical data types for Oracle***

Oracle Type	Bdp Type	BdpSubType	System.Type
CHAR	String	stFixed	String
NCHAR	String	stFixed	String
VARCHAR	String	NA	String
NVARCHAR	String	NA	String
VARCHAR2	String	NA	String
NVARCHAR2	String	NA	String
NUMBER	Decimal	NA	Decimal
DATE	Date	NA	DateTime
BLOB	Blob	stHBinary	Byte[]
CLOB	Blob	stHMemo	Char[]
LONG	Blob	stMemo	Char[]
LONG RAW	Blob	stBinary	Byte[]
BFILE	Blob	stBFile	Char[]

ROWID	String	NA	String
-------	--------	----	--------

# BDP.NET Component Designers

---

Almost all distributed applications revolve around reading and updating information in databases. Different applications you develop using ADO.NET requires different requirements for working with data. For instance, you might develop an application that simply displays data on a form. Or, you might develop an application that provides a way to share data information with another company. No matter what your intent is, you need to have an understanding of certain fundamental concepts about data approach in ADO.NET.

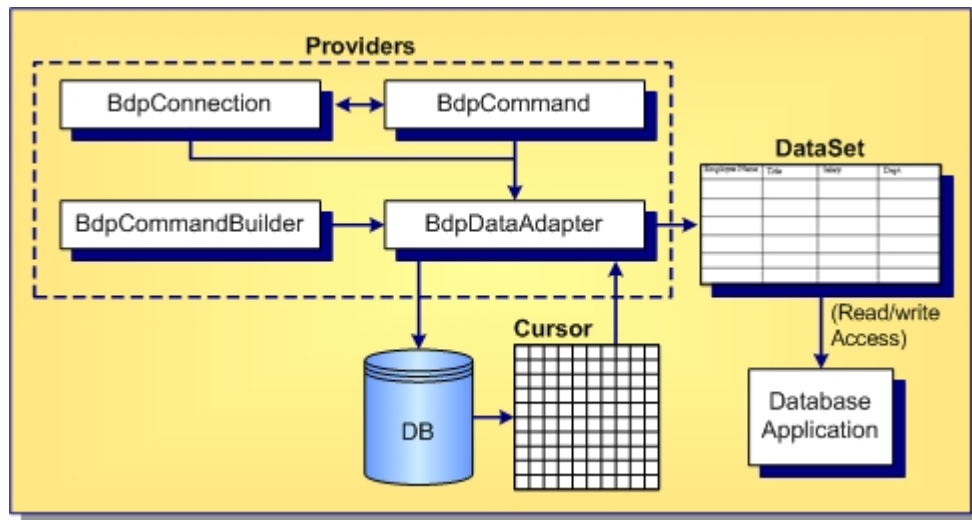
Delphi 8 for .NET provides BDP.NET component designers to allow data integration in distributed, scalable applications. By providing a number of these designers, you can work efficiently to access, expose, and edit data. These designers let you browse and edit database server-specific schema objects like tables, fields, triggers, and indexes, extensively allow you to use these existing tools to connect and perform the same type of functionality to a number of industry standard databases, and many others.

This section includes:

- Component Designer Relationships
- Connection Editor
- Command Text
- Generate Datasets
- Configure Data Adapter
- Data Explorer

## Component Designer Relationships

---



The major components of the database component designers include a *Connection Editor* to define a live connection to a data source, a *Command Text Editor* to construct command text for command components, a *Configure Data Adapter* to set up commands for a data adapter, and a *Generate Dataset* to build custom datasets. Additionally, you can use the *Data Explorer* to browse database server-specific schema objects and use drag and drop techniques to automatically populate data from a data source to your Delphi for .NET project.

### Connection Editor

---

Delphi 8 for .NET provides a *Connection Editor* designer that lets you select a connection configuration or edit the named connections that are stored in an XML configuration file. This editor lets you add, delete, and test your connection. Display this dialog by dropping the BdpConnection component from the *Tool Palette* onto the form. Click the component designer verb at the bottom of the *Object Inspector* to launch the dialog .

### Command Text Editor

---

Delphi 8 for .NET provides a *Command Text Editor* designer lets you construct the command text for command components that have a CommandText property. A multi-line editing control in the editor lets you manually edit the command or build the command text by selecting tables and columns. Display this dialog by dropping the BdpCommand component from the *Tool Palette* onto the form. Click the designer verb at the bottom of the *Object Inspector* to launch the dialog.

To populate the Tables and Columns list boxes with items and build SQL statements, you must have defined a live BdpConnection. Otherwise, data will not be retrieved.

## Generate DataSets

---

Delphi 8 for .NET provides a *Generate Dataset* designer that lets you build a Dataset. Some of the advantages for using this tool include strong typing, cleaner code, and the ability to use code completion. A Dataset is first derived from the base Dataset class and then uses information in an XML Schema file (an .xsd file) to generate a new class. Information from the schema (tables, columns, and so on) is generated and compiled into this new dataset class as a set of first-class objects and properties. Display this dialog by dropping a BdpDataAdapter component from the *Tool Palette* onto the form. Click the component designer verb at the bottom of the *Object Inspector*. If this component is not displayed, choose Component ► Installed .NET Components to add it to the *Tool Palette*.

## Configure Data Adapter

---

Delphi 8 for .NET provides a *Configure Data Adapter* designer which lets you configure SQL statements or stored procedures that are invoked to read or write to a database. Data Adapters are an integral part of the ADO.NET managed providers, which are a set of objects used to communicate between a data source and a dataset. Essentially, Adapters are used to exchange data between a data source and a dataset. This means reading data from a database into a dataset, and then writing changed data from the dataset back to the database. A Data Adapter can move data between any source and a dataset. Display this dialog by dropping a BdpDataAdapter component from the *Tool Palette* onto the form. Click the component designer verb at the bottom of the *Object Inspector*.

To populate the Tables and Column list boxes with items and build SQL statements, you must have defined a live BdpConnection. Otherwise, data will not be retrieved.

## Data Explorer

---

Delphi 8 for .NET provides a hierarchical database browser and editing tool which lets you browse database server-specific schema objects including tables, fields, stored procedure definitions, triggers, and indexes. The *Data Explorer* can also be extended to connect to other popular industry standard databases. Once connected, you can view schema objects from other databases and modify connections. Display this tool by selecting View ► Data Explorer. Additionally, the *Data Explorer* can be used to drag and drop data from a data source to any Delphi 8 for .NET project. This feature is especially useful if you have a long connection string and want to automatically populate the connection string to avoid inputting the string improperly.



# Delphi 8 for .NET Database Technologies

---

Delphi 7 provided a number of database technologies to allow you to accomplish different types of database tasks. Some technologies, like BDE, were included mainly for backward compatibility to older database systems. Others, like dbExpress, were included as a way to create cross-platform applications. The move to the .NET Framework creates new opportunities and challenges for the database developer. In most cases, the .NET Framework's ADO.NET providers or Borland's BDP.NET providers will solve the challenges of creating new database applications. Sometimes, however, there is little to be gained by creating an entirely new application when an existing application will serve your purposes. That's why Borland is providing continued support for existing Delphi database technologies, in addition to giving you easy access to the newer BDP.NET and ADO.NET.

Delphi 8 for .NET provides a migration path from Delphi database technologies running strictly on Win32 clients to the .NET Framework. In addition to being able to build new database applications using ADO.NET and BDP.NET, you can migrate existing database applications to take advantage of .NET capabilities. The Delphi database technologies now supported by Delphi 8 for .NET include:

- TADONETConnector
- dbExpress.NET
- DataSnap .NET Client (DCOM)
- IBX.NET (InterBase for .NET)
- BDE.NET

## Building .NET Applications with TADONETConnector

---

Delphi 8 for .NET includes a new descendant of TDataSet, called TADONETConnector. This class provides the means to connect to any ADO.NET or BDP.NET data adapter. It offers perhaps the easiest migration path of any existing applications that use one of the other TDataSet descendents, such as TADODataset. The class is easily implemented and requires setting a minimal number of properties to make a connection to an existing BDP.NET or ADO.NET data adapter.

## Building .NET Applications with dbExpress.NET

---

Delphi 8 for .NET includes a .NET version of dbExpress, called dbExpress.NET. This set of components provide comparable functionality as the Delphi 7 dbExpress, but updated to run on Java on the .NET Framework. dbExpress.NET provides the same lightweight client capability and unidirectional dataset that is available in Delphi 7.

## **Building .NET Applications with IBX.NET**

---

Delphi 8 for .NET provides you with access to InterBase databases, by way of InterBase Express controls, in addition to the standard BDP.NET data adapter or the .NET Framework's ADO.NET providers. IBX.NET controls allow you to connect to InterBase databases, access tables and datasets,

## **Building .NET Applications with the DataSnap .NET Client (DCOM)**

---

Delphi 8 for .NET provides the means to use the Midas (DCOM) client to connect to databases in three-tier applications.

## **Building .NET Applications with BDE.NET**

---

You can connect your Delphi 8 for .NET database applications to BDE-supported databases, such as Paradox and dBase.

# Getting Started with InterBase Express














---

InterBase Express (IBX) is a set of data access components that provide a means of accessing data from InterBase databases. The InterBase Administration Components, which require InterBase 6, are described after the InterBase data access components.

## IBX components

---

The following components are located on the InterBase tab of the component palette.

	TIBTable
	TIBQuery
	TIBStoredProc
	TIBDatabase
	TIBTransaction
	TIBUpdateSQL
	TIBDataSet
	TIBSQL
	TIBDatabaseInfo
	IBSQLMonitor
	TIBEvents
	TIBExtract
	TIBClientDataSet

Though they are similar to BDE components in name, the IBX components are somewhat different. For each component with a BDE counterpart, the sections below give a discussion of these differences.

There is no simple migration from BDE to IBX applications. Generally, you must replace BDE components with the comparable IBX components, and then recompile your applications. However, the speed you gain, along with the access you get to the powerful InterBase features make migration well worth your time.

### IBDatabase

Use a IBDatabase component to establish connections to databases, which can involve one or more concurrent transactions. Unlike BDE, IBX has a separate transaction component, which allows you to separate transactions and database connections.

To set up a database connection:

1. Drop an IBDatabase component onto a form or data module.
2. Fill out the DatabaseName property. For a local connection, this is the drive, path, and filename of the database file. Set the Connected property to true.
3. Enter a valid username and password and click OK to establish the database connection.

**Warning:** Tip: You can store the username and password in the IBDatabase component's Params property by setting the LoginPrompt property to false after logging in. For example, after logging in as the system administrator and setting the LoginPrompt property to false, you may see the following when editing the Params property:

```
user_name=sysdba  
password=masterkey
```

### IBTransaction

Unlike the Borland Database Engine, IBX controls transactions with a separate component, TIBTransaction. This powerful feature allows you to separate transactions and database connections, so you can take advantage of the InterBase two-phase commit functionality (transactions that span multiple connections) and multiple concurrent transactions using the same connection.

Use an IBTransaction component to handle transaction contexts, which might involve one or more database connections. In most cases, a simple one database/one transaction model will do.

To set up a transaction:

1. Set up an IBDatabase connection as described above.
2. Drop an IBTransaction component onto the form or data module
3. Set the DefaultDatabase property to the name of your IBDatabase component.
4. Set the Active property to true to start the transaction.

### IBX dataset components

There are a variety of dataset components from which to choose with IBX, each having their own characteristics and task suitability:

## IBTable

Use an IBTable component to set up a live dataset on a table or view without having to enter any SQL statements.

IBTable components are easy to configure:

1. Add an IBTable component to your form or data module.
2. Specify the associated database and transaction components.
3. Specify the name of the relation from the TableName drop-down list.
4. Set the Active property to true.

## IBQuery

Use an IBQuery component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables.

IBQuery components provide a read-only dataset, and adapt well to the InterBase client/server environment. To set up an IBQuery component:

1. Set up an IBDatabase connection as described above.
2. Set up an IBTransaction connection as described above.
3. Add an IBQuery component to your form or data module.
4. Specify the associated database and transaction components.
5. Enter a valid SQL statement for the IBQuery's SQL property in the String list editor.
6. Set the Active property to true

## IBDataSet

Use an IBDataSet component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables. IBDataSet components are similar to IBQuery components, except that they support live datasets without the need of an IBUpdateSQL component.

The following is an example that provides a live dataset for the COUNTRY table in employee.gdb:

1. Set up an IBDatabase connection as described above.
2. Specify the associated database and transaction components.
3. Add an IBDataSet component to your form or data module.
4. Enter SQL statements for the following properties:

---

SelectSQL	SELECT Country, Currency FROM Country
-----------	---------------------------------------

RefreshSQL	SELECT Country, Currency FROM Country WHERE Country = :Country
------------	--

ModifySQL	UPDATE Country SET Country = :Country, Currency = :Currency WHERE Country = :Old_Country
-----------	---

---

DeleteSQL	DELETE FROM Country WHERE Country = :Old_Country
InsertSQL	INSERT INTO Country (Country, Currency) VALUES (: Country, :Currency)

---

1. Set the Active property to true.
- 2.

**Note: Note:** Parameters and fields passed to functions are case-sensitive in dialect 3. For example,

```
FieldByName (EmpNo)
```

would return nothing in dialect 3 if the field was 'EMPNO'.

### IBStoredProc

Use IBStoredProc for InterBase executable procedures: procedures that return, at most, one row of information. For stored procedures that return more than one row of data, or "Select" procedures, use either IBQuery or IBDataSet components.

### IBSQL

Use an IBSQL component for data operations that need to be fast and lightweight. Operations such as data definition and pumping data from one database to another are suitable for IBSQL components.

In the following example, an IBSQL component is used to return the next value from a generator:

1. Set up an IBDatabase connection as described above.
2. Put an IBSQL component on the form or data module and set its Database property to the name of the database.
3. Add an SQL statement to the SQL property string list editor, for example:

```
SELECT GEN_ID(MyGenerator, 1) FROM RDB$DATABASE
```

### IBUpdateSQL

Use an IBUpdateSQL component to update read-only datasets. You can update IBQuery output with an IBUpdateSQL component:

1. Set up an IBQuery component as described above.
2. Add an IBUpdateSQL component to your form or data module.
3. Enter SQL statements for the following properties: DeleteSQL, InsertSQL, ModifySQL, and RefreshSQL.
4. Set the IBQuery component's UpdateObject property to the name of the IBUpdateSQL component.

5. Set the IBQuery component's Active property to true.

### **IBSQLMonitor**

Use an IBSQLMonitor component to develop diagnostic tools to monitor the communications between your application and the InterBase server. When the TraceFlags properties of an IBDatabase component are turned on, active IBSQLMonitor components can keep track of the connection's activity and send the output to a file or control.

A good example would be to create a separate application that has an IBSQLMonitor component and a Memo control. Run this secondary application, and on the primary application, activate the TraceFlags of the IBDatabase component. Interact with the primary application, and watch the second's memo control fill with data.

### **IBDatabaseInfo**

Use an IBDatabaseInfo component to retrieve information about a particular database, such as the sweep interval, ODS version, and the user names of those currently attached to this database.

For example, to set up an IBDatabaseInfo component that displays the users currently connected to the database:

1. Set up an IBDatabase connection as described above.
2. Put an IBDatabaseInfo component on the form or data module and set its Database property to the name of the database.
3. Put a Memo component on the form.
4. Put a Timer component on the form and set its interval.
5. Double click on the Timer's OnTimer event field and enter code similar to the following:

```
Mem1.Text := IBDatabaseInfo.UserNames.Text;    // Delphi example
Mem1->Text = IBDatabaseInfo->UserNames->Text; // C++ example
```

### **IBEvents**

Use an IBEvents component to register interest in, and asynchronously handle, events posted by an InterBase server.

To set up an IBEvents component:

1. Set up an IBDatabase connection as described above.
2. Put an IBEvents component on the form or data module and set its Database property to the name of the database.
3. Enter events in the Events property string list editor, for example:

```
IBEvents.Events.Add('EVENT_NAME');    // Delphi example
```

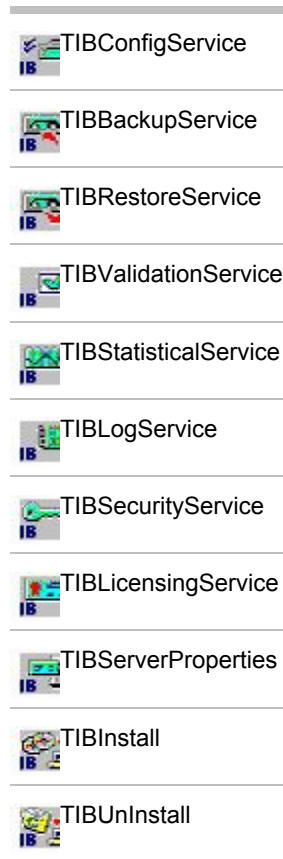
```
IBEvents->Events->Add("EVENT_NAME"); // C++ Example
```

1. 4. Set the Registered property to true.
- 2.

### InterBase Administration Components

If you have InterBase 6 installed, you can use the InterBase 6 Administration components, which allow you to use access the powerful InterBase Services API calls.

The components are located on the InterBase Admin tab of the IDE and include:



Note: You must install InterBase 6 to use these features.

#### IBConfigService

Use an TIBConfigService object to configure database parameters, including page buffers, async mode, reserve space, and sweep interval.

#### IBBackupService



Use an `TIBBackupService` object to back up your database. With `IBBackupService`, you can set such parameters as the blocking factor, backup file name, and database backup options.

#### **IBRestoreService**

Use an `TIBRestoreService` object to restore your database. With `IBRestoreService`, you can set such options as page buffers, page size, and database restore options.

#### **IBValidationService**

Use an `TIBValidationService` object to validate your database and reconcile your database transactions. With the `IBValidationService`, you can set the default transaction action, return limbo transaction information, and set other database validation options.

#### **IBStatisticalService**

Use an `TIBStatisticalService` object to view database statistics, such as data pages, database log, header pages, index pages, and system relations.

#### **IBLogService**

Use an `TIBLogService` object to create a log file.

#### **IBSecurityService**

Use an `TIBSecurityService` object to manage user access to the InterBase server. With the `IBSecurityService`, you can create, delete, and modify user accounts, display all users, and set up work groups using SQL roles.

#### **IBLicensingService**

Use an `TIBLicensingService` component to add or remove InterBase software activation certificates.

#### **IBServerProperties**

Use an `TIBServerProperties` component to return database server information, including configuration parameters, and version and license information.

#### **IBInstall**

Use an `TIBInstall` component to set up an InterBase installation component, including the installation source and destination directories, and the components to be installed.

#### **IBUnInstall**

Use an `TIBUnInstall` component to set up an uninstall component.

# Building VCL.NET Applications

---

VCL.NET gives you the capability to provide your Delphi VCL applications and components to Microsoft .NET Framework users. VCL.NET is an extended set of the VCL components that provide the means to quickly and easily build advanced applications in Delphi. Now, with Delphi 8 for .NET you get all of the benefits of the .NET Framework along with the ease-of-use and powerful component-driven application development you've come to expect from Delphi.

Delphi 8 for .NET provides distinct application types for your use: you can create VCL.NET form applications that run on the .NET Framework, but which use VCL.NET components and controls; you can create .NET Windows Forms applications that use the underlying .NET Framework and .NET controls while offering Delphi 8 for .NET code behind; you can create powerful ASP.NET applications that use the underlying .NET Framework, ASP.NET controls, and also offer Delphi 8 for .NET code behind. The following topics provide more information on how to take advantage of the new VCL.NET provisions in Delphi 8 for .NET.

# VCL.NET Overview

---

VCL.NET is the programming framework for building Delphi 8 for .NET applications using VCL components. Delphi 8 for .NET and VCL.NET are intended to help users leverage the power of Delphi when writing new applications, as well as for migrating existing Win32 applications to the .NET Framework.

The aim with these technologies is to allow a Delphi developer to move to .NET, taking their Delphi skills and much of their current Delphi source code with them. Delphi 8 for .NET supports Microsoft .NET Framework development with the Delphi language and both Visual Component Library (VCL) for .NET controls and Windows Forms controls. Delphi 8 for .NET ASP.NET also supports WebForms and SOAP/XML Web Services application development.

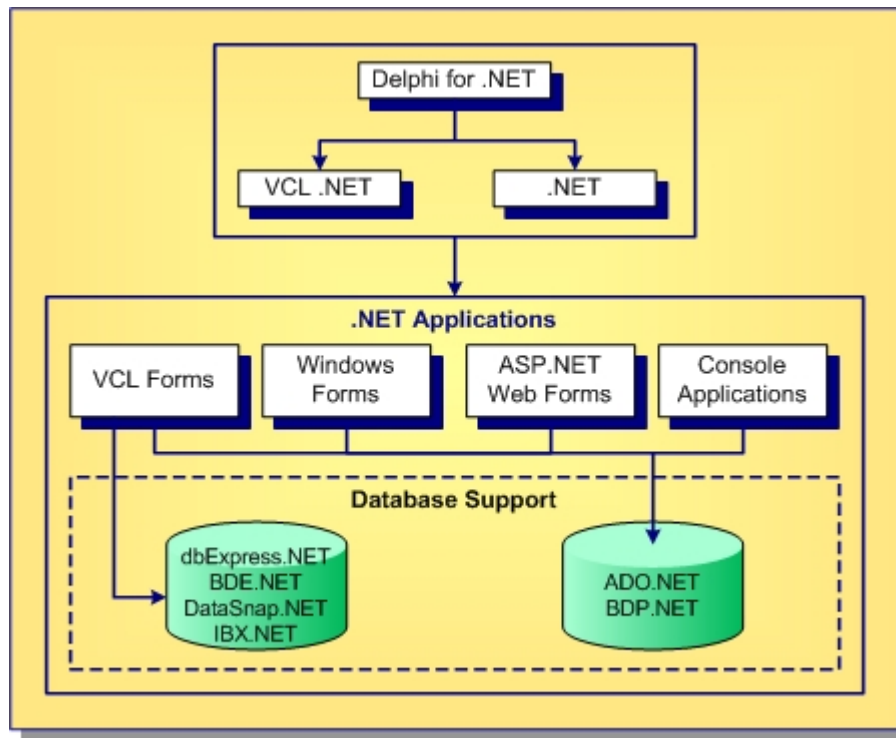
VCL.NET is a large subset of the most common classes in VCL for Win32. Because the .NET Framework was architected to accommodate any .NET-compliant language, in many cases Delphi source code that operates on Win32 VCL classes and functions recompiles with minimal changes on .NET. In some cases, the code recompiles with no changes at all. VCL.NET is a large subset of VCL, therefore it supports many of the existing VCL classes. However, source code that calls directly to the Win32 API requires source code changes. Also, dependent third-party Win32 VCL controls need to be available in .NET versions as well for compatibility.

This section introduces:

- VCL.NET architecture.
- VCL.NET components.
- Borland.VCL namespace.
- VCL.NET application deployment.

## What is VCL?

---



VCL is an acronym for the Visual Component Library, a set of visual components for building Windows applications in the Delphi language. VCL.NET is the same library of components, updated for use in the .NET Framework. As you can see in the preceding illustration, VCL.NET and the .NET Framework coexist within Delphi 8 for .NET. Both frameworks (VCL.NET and .NET) provide components and functionality that allow you to build .NET applications:

- VCL.NET provides the means to create VCL Forms applications, which are Delphi forms that are .NET-enabled, and use VCL.NET components.
- VCL.NET provides VCL non-visual components which have been .NET-enabled to access databases and also allows you to access databases by way of ADO.NET and BDP.NET providers.
- .NET provides the means to build .NET Windows Forms, Web Forms, and Console applications, using .NET components, with Delphi code behind.
- .NET provides data access components that allow you to access databases by way of ADO.NET.

You can build VCL Forms applications using VCL.NET components or Windows Forms applications using .NET components. You can also build ASP.NET Web Forms applications using either VCL.NET components or .NET components.

## **The Relationship between VCL.NET and the .NET Framework**

---

It's important to understand the relationship between VCL.NET and the .NET Framework. The .NET Framework provides a library of components, classes, and low-level functionality that supports a more transparent type of application development, particularly for Web applications. The concept of transparency, in this context, means that the framework manages much of the common functionality, from the display of buttons to remoting functionality, without regard to the underlying implementation language. It would be an understandable mistake to assume that VCL.NET is subordinate to the .NET Framework, and yet, it is more accurate to view VCL.NET and the .NET Framework as functionally equivalent. Like the .NET Framework, VCL.NET provides libraries of components, controls, classes, and lower-level functionality that help you build Windows Forms, Web Forms, and Console applications that run on the current Windows .NET Framework platform.

Is VCL.NET a replacement for the .NET Framework?

Yes and No. You will still need the .NET runtime to use VCL.NET, but you can build complete applications using VCL.NET components that will run on .NET platform.

Can you build Delphi 8 for .NET applications without using VCL.NET?

Yes, you can create Windows Forms, Web Forms, and Console applications using Delphi 8 for .NET code.

What it really means for you, as a developer, is that you can use Delphi 8 for .NET to create powerful .NET applications using .NET components, or that you can use VCL.NET components that have been migrated from the Delphi VCL. If you have existing Delphi VCL applications that you want to run on Windows XP or other platforms that support the .NET Framework, you can easily port those applications by using Delphi 8 for .NET.

## **VCL.NET Components**

---

As a Visual Component Library for .NET, the heart of VCL.NET consists of a set of components, visual and non-visual. VCL.NET continues to build on the concept of constructing applications visually, thus eliminating as much hand-coding as possible.

### **Visual Components**

Delphi 8 for .NET provides a set of visual components, or controls, that you can use to build your applications. In addition to the common controls such as buttons, text boxes, radio buttons, and check boxes, you can also find grid controls, scroll bars, spinners, calendar objects, a full-featured menu designer, and more. There is, however, a major difference in the way these controls are represented as opposed to the way visual components are represented by other frameworks, such as the .NET Framework.

In the Code Editor of Delphi 8 for .NET you will not see a code representation of your visual components. In IDEs for other languages, such as C# or Java, it is common to see code-centric representations of

forms and other visual components. These representations include physical definitions, such as size, height, and other properties, as well as constructors and destructors for the components.

Delphi 8 for .NET is a resource-centric type of system, which means that the primary code-behind representations are of event handlers that you fill in with your program logic. Visual components are declared and defined in textual files, given the extension of *.dfm* (for Delphi Forms) or *.nfm* (for Delphi 8 for .NET forms). The *.nfm* files are created by Delphi 8 for .NET as you design your VCL Forms on the Forms Designer, and are listed in the resource list in the Project Manager for the given project.

## Non-Visual Components

You can use non-visual components to implement functionality that is not necessarily exposed visually. For example, you can accomplish data access by using non-visual components like the BDP.NET components, which provide database connectivity and dataset access. Even though these components don't have a visual runtime behavior, they are represented by components in the Tools Palette at designtime. VCL.NET provides a variety of non-visual components for data access, server functions, and more.

## Classes

VCL.NET also includes classes that are not components. In other words, they descend from *TObject* but not from *TComponent*, and are typically used for accessing system objects such as a file, or for transient tasks. Many of these tasks have parallel classes in the .NET Framework.

Classes, whether components or not, contain a number of characteristics, which can be categorized as properties, methods, and events. In addition, most elements of a class are of a particular type, for instance, simple data types like integers and strings, or complex data types, such as specific class types. These elements, properties, methods, events, and types, allow you to make controlled modifications to your classes, generally by way of the Object Inspector in the IDE. The use of these elements is just one more way to simplify application development and to encourage the visual approach.

## VCL Forms

Delphi 8 for .NET provides a combined *Code Editor* and *VCL Forms Designer*, which gives you access to a rich set of VCL.NET controls and non-visual components for drag-and-drop user interface design, code-behind code generation, integrated debugging, and project building capabilities. VCL.NET Forms are descendants of *TObject* and accept any VCL.NET controls or VCL controls that have been updated to conform to .NET Framework requirements.

## Windows Forms

Windows Forms and their available components are provided by the .NET Framework. Delphi 8 for .NET includes an integrated *Code Editor* and *Windows Form Designer* that allows you to build fully-compliant .NET Windows applications from the ground up, using only .NET components. In Delphi 8

for .NET the Windows Form object is created as a `TWinForm`, which is a descendant of the `TForm` class and accepts any fully-compliant .NET components.

## Web Forms

Web Forms and their available components are provided by the .NET Framework. Delphi 8 for .NET includes an integrated *Code Editor* and *Web Form Designer* that allows you to build fully-compliant .NET ASP.NET applications from the ground up, using only .NET components. In Delphi 8 for .NET the Web Form object accepts any fully-compliant .NET components. Delphi 8 for .NET also includes a full set of data aware database components for ASP.NET.

## Database Providers

Most of the current VCL database access technology is also being developed for .NET in addition to full support for ADO.NET including the Borland Data Providers. You are able to use such .NET-enabled data access technologies as dbExpress.NET, BDE.NET, the DataSnap Client for .NET, and IBX.NET. Each of these technologies is exposed as a set of non-visual components, but can also be instantiated in code.

## Object Pascal

All code-behind code generation in Delphi 8 for .NET results in Object Pascal, more commonly known as Delphi code. When using Delphi 8 for .NET you will see code files created with the extension *.pas*, which indicates a **pascal** file.

## Borland.VCL Namespace

---

VCL.NET classes are found under the `Borland.Vcl` namespace. Database-related classes are in the `Borland.Vcl.DB` namespace. Runtime library classes are in the `Borland.Vcl.Rtl` namespace.

If you are a user of earlier Delphi versions, you will find that the unit files with which you are familiar, have been bundled in corresponding `Borland.Vcl` namespaces. In some cases, units have been moved, however, every attempt has been made to identify the namespaces in meaningful ways, and in ways that will assist you in finding the functionality you want.

Source files for all of the Delphi 8 for .NET objects are available in the `c:\Program Files\Borland\BDS\12.0\Source` subdirectory.

## Porting Delphi Applications to Delphi 8 for .NET

---

If you have existing applications written using an earlier version of Delphi, you might want to port them to .NET. In most cases, this will likely be easier than writing the applications over from scratch. Because Delphi 8 for .NET takes advantage of significant structural elements in the .NET Framework, you will need to perform some manual porting tasks to make your applications run. For example, the .NET Framework does not support pointers in safe code. So, any instances of `pChar` or pointer variables will

need to be revised to one of the .NET types. Many Delphi objects have been already updated to accommodate these type restrictions, but your code may include references to pointers or unsupported types. For more information, refer to the Porting information in this Help system, and to the Language Guide.

## **Importing .NET Components for Use in VCL.NET Applications**

---

Delphi 8 for .NET provides the *.NET Import Wizard*, which helps you import .NET controls into VCL.NET units and packages. You can wrap all .NET components, for instance, those from the System.Windows.Forms assembly, in ActiveX wrappers that can be deployed on VCL.NET applications. Once you have imported the .NET components of your choice, you can add a completed package file containing the units for each component to the Tool Palette. You can also view and modify the individual unit files, if necessary. These files can be useful reference material when you are writing your own custom components.



# Porting VCL Applications

---

When porting VCL applications from Delphi 7 to Delphi 8 for .NET, there are a number of issues you need to consider. Along with basic language elements that need to be replaced or modified, there are strategies that you should follow to make sure that you port your applications fully and reliably.

This topic describes

- General language issues in porting.
- Porting web service client applications.

## General Language Issues

---

With the move to the .NET Framework, a number of language issues have come into play when it comes to porting Delphi 7 applications to Delphi 8 for .NET. The .NET Framework, for instance, considers pointers to be *unsafe* and so does not consider applications that use pointers to fall into the category of *managed* code. In order to be compliant with the framework, you will need to make adjustments to your applications to avoid or circumvent the use of pointers, the pChar type, and other language-specific elements.

In addition, there are critical issues in working with the Win32 API, using crackers, migrating char types, and other topics.

## New Language Features

---

Several new features have been added to the Delphi language, to support programming concepts and features of the .NET platform and the Common Language Specification (CLS):

- Partitioning code into namespaces
- New visibility specifiers for class members
- Class static methods, properties, and fields
- Class constructors
- Nested type declarations within classes
- Sealed classes
- Final virtual methods
- Operator overloads in classes
- .NET attributes
- Class helper syntax

Programming in the garbage collected environment of .NET brings a number of new issues related to allocating and disposing objects. These issues are discussed in Memory Management Issues on the .NET Platform.

## **Porting Web Service Client Applications**

---

The .NET Framework employs a major architectural shift in how it handles web services and web service clients. As such, your existing web service client applications need to be modified to operate on the framework. Delphi 8 for .NET does not support the RIO components, and uses a more transparent .NET approach to managing web service client applications. You will need to eliminate RIO components and modify the way you access WSDL documents.

# Building Reports in Delphi 8 for .NET

---

Delphi 8 for .NET ships with the Rave Reports from Nevrona. Using Rave, you can build full-featured reports for your Delphi 8 for .NET applications, and you can create solutions that include reporting capabilities which can be used and customized by your own customers. In addition, the ComponentOne tools that ship with Delphi 8 for .NET also include components for creating and generating reports, which can be integrated into your applications.

# Using Rave Reports in Delphi 8 for .NET

---

The Delphi 8 for .NET environment supports the integration of certain Rave Reports objects in your applications. This integration allows you to create a report using the Rave Reports Designer or to drag-and-drop Rave Reports ActiveX components directly onto your Windows Forms and Web Forms in the Delphi 8 for .NET Designer. You can enable your application users to create and display their own reports, or to display existing reports they have built using Rave Reports. The Delphi 8 for .NET integration with Rave Reports allows you to:

- Include new report objects in projects.
- Drag-and-drop Rave Reports ActiveX objects onto Windows Forms and Web Forms.

## Creating New Reports in Delphi 8 for .NET

---

You can include reports in Delphi 8 for .NET just as you would other 3rd-party components. The report is stored as a separate Rave Report object. You can reference the report in any number of applications that have a need to call or generate that report. When you create a new application, you can include the report object by adding a reference to it in the Project Manager. Rave Reports also provide the capability to connect your report object to a datasource, which allows the report to be built dynamically based on up-to-date database information. For more information about Rave Reports and about connecting to data sources, refer to the Rave Reports online help.

## Using Rave Reports ActiveX Components

---

You can drag-and-drop any ActiveX objects that are exposed by Rave Reports to your applications. The Delphi 8 for .NET tool palette provides a list of any available ActiveX objects. Just drag-and-drop the objects you want onto a Windows Form or a Web Form during design. Fill in the appropriate properties and modify any code in the *Code Editor*. You may need to reset your .NET components and to select the ActiveX components you want displayed, by selecting them from the *Installed .NET Components* dialog.

# Procedures

## Getting Started

---

# Adding Components to a Form

---

## To add components to a form

---

1. On the *Tool Palette*, select a visual or nonvisual component.
2. Double-click the component to place it on the form, or click the component once and then click an area within the form where you want the component to appear. Alternatively, you can click and drag the component onto the form.

If you add a nonvisual component to the form, the component tray appears at the bottom of the designer surface.

3. Repeat steps 1 and 2 to add additional components.
4. Use the dotted grid on the form to align your components.

# Adding References

---

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

## To add references

---

1. From the main menu, choose Project ► Add References. The *Add References* dialog appears.
2. Select either a legacy COM server or ActiveX control to integrate into your managed application.
3. Click *Add Reference*. The reference is added to the text box.
4. Click *OK*.

**Tip:** Alternatively, in the *Project Manager*, right-click the *Reference* folder and choose *Add Reference*.



# Adding and Removing Files

---

## To add a file to a project

---

1. Choose Project ► Add to Project.

The *Add to Project* dialog box appears.

2. Select a file to add and click *Open*.

The file appears below the Project.exe node of the *Project Manager*.

## To remove a file from a project

---

1. Choose Project ► Remove From Project.

A *Remove From Project* dialog box appears.

2. Select the file(s) you want to remove and click *OK*.

# Adding Templates to the Object Repository

---

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

## To add a template to the Object Repository

---

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

# Copying References to a Local Path

---

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

## To a copy reference to a local path

---

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

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

# Creating a Component Template

---

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

## To create a component template

---

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

Gray handles appear at the corners of each selected component.

4. Choose Component ► Create Component Template.

The *Create Component Template* dialog box is displayed.

5. Specify a name, a *Tool Palette* category, and an icon for the template.
6. Click OK.

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

## To use a component template

---

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

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

## To delete a component template

---

1. On the *Tool Palette*, right-click the component template to display a context menu.

2. Choose the Delete [template name] Button command.

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

# Creating a Project

---

## To add a new project

---

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

## To add an existing project

---

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

# Customizing the Form

---

## To customize the form

---

1. Choose Tools ► Options.
2. From the *Options* dialog box, scroll down to *Windows Forms Designer*.
3. Enable or disable the snap to grid and show grid features by selecting and deselecting the check boxes.

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

# Customizing Toolbars

---

## To arrange your toolbars

---

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

## To delete icons from the toolbar

---

1. Choose View ► Toolbars ► Customize.
2. From the toolbar, not the *Customize* dialog box, drag the tool from the toolbar until its icon displays an X and release the mouse button.

## To add icons to the toolbar

---

1. Choose View ► Toolbars ► Customize.
2. Select the *Commands* tab.
3. In the *Categories* list, select a category to view its tool icons.
4. From the *Commands* list, drag the selected icon and command onto the toolbar of your choice.



# Customizing the Tool Palette

---

## To arrange individual components

---

1. Select the component.
2. Hold and drag the component anywhere within the *Tool Palette* to rearrange it.

## To arrange an entire category of components

---

1. Grab an entire category and move it anywhere in the *Tool Palette*.
2. Release your mouse to place the category in the desired location.

## To add additional categories

---

1. Right-click the *Tool Palette*.
2. Scroll down to *Add New Category* option.

# Docking Tool Windows

---

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

## To use Auto-Hide to hide your tools

---

1. Click the push pin in the upper right corner of the tool window.

The tool window is replaced by one or more tabs at the outer edge of the IDE window.

2. To display the tool window, position the cursor over the tab.

The tool window slides into view.

3. To slide the tool window out of view, move the cursor away from the tool window.

4. To redock the tool window, click the push pin until it points down.

## To dock the tools with one another

---

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

## To undock the tools from one another

---

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

# Filtering Searched Components

---

## To filter searched components

---

1. On the *Tool Palette*, start typing the name of the component that you want to find in the search text box.

A list of component names that match what you are typing appears in the *Tool Palette*.

2. Double-click the component to add it to your form.

# Exploring .NET Assembly Metadata


---

The Delphi 8 for .NET IDE allows you to open and explore the namespaces and types contained within a .NET assembly. The assembly metadata is displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the namespaces and types within the assembly. The right pane displays specific information on the selected item in the tree. The *Call Graph tab* shows you a list of the methods called by the selected method, as well as a list of the methods that call the selected method.

## To Inspect a .NET Assembly

---

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

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

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

## Using the Call Graph Tab

---

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

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

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

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

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

# Exploring Windows Type Libraries


---


The Delphi 8 for .NET IDE allows you to open and inspect the interfaces and other types contained within a Windows type library. The type library contents are displayed in a Windows Explorer-style presentation, with a left pane containing a tree of the interface and type definitions within the type library. The right pane displays specific information on the selected item in the tree. The *type library explorer* can open a .TLB file, as well as OCX controls, and .DLL and .EXE files that have type libraries as embedded resources.

## To Inspect a Windows Type Library

---

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

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

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

# Installing Custom Components

---

If you create custom components or obtain them from a third-party vendor, you can install them on the *Tool Palette* and then use them in your applications.

## To install custom components

---

1. Choose Component ➤ Installed .NET Components.
2. Click *Select an Assembly*.
3. Navigate to the folder that contains the assembly which contains the components to install and select it.

Alternatively, you can enter the name of the full path to the assembly in the *File Name* field.

4. Click *Open*.

The *Installed .NET Components* dialog displays the components from the assembly.

5. Verify that the components you want to install on the *Tool Palette* are checked.
6. Click *OK*.

# Renaming Files using the Project Manager

---

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

## To rename a file

---

1. In the *Project Manager*, right-click the file that you want to rename. Its context menu is displayed.
2. Choose *Rename*.
3. Enter the new name for the file.

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

# Saving Desktop Layouts

---

You can switch between multiple desktop layouts whenever you want. Choose a layout from the drop-down pick list box located on the main menu. Additionally, you can save your desktop or debug desktop layouts as default.

## To save a desktop layout

---

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

## To set a debug desktop layout

---

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



# Setting Component Properties

---

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

## To set component properties

---

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

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

# Setting Dynamic Properties

---

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

## To set a dynamic property in the Object Inspector

---

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

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

3. Click the ellipsis button next to *(Advanced)* to display the *Dynamic Properties* dialog box.  
This dialog lists all of the properties that can be stored in the configuration file.

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

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

Delphi 8 for .NET creates an XML file named *app.config* (for a Windows application) or *Web.config* (for a Web application) in the project directory. This file lists the dynamic properties and their current values.

7. Compile the application.

Delphi 8 for .NET creates a file named *projectname.exe.config* (for a Windows application) or *projectname.dll.config* (for a Web application) in the same directory as the application's executable or DLL file.

## To change a dynamic property value in the configuration file

---

1. In the directory that contains the application's executable or DLL file, locate the configuration file.

2. Open the file in a text editor.
3. Locate the add key= statement for the property to be changed and edit the value.
4. Save your changes and close the file.

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

# Setting Project Options

---

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

## To change compiler options

---

1. Choose Project ► Options.

The *Options* dialog box appears.

2. Select *Compiler* and set your options to modify how you want your program to compile.

## To change application options

---

1. Choose Project ► Options.

The *Options* dialog box appears.

2. Select *Application* and specify a title and extension for your application.

## To change debugger options

---

1. Choose Project ► Options.

The *Options* dialog box appears.

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

# Setting Properties and Events

---

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

## To set object properties

---

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

## To set an event handler

---

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

# Setting Tool Preferences

---

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

## To set tool preferences

---

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

# Using To-Do Lists

---

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

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

---

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

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

---

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

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

## To mark a to-do list item as completed

---

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

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

## To filter the items in a to-do list

---

1. Choose View ► To-Do List.

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

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

## To delete an item from a to-do list

---

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

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



# Writing Event Handlers

---

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

## To write an event handler

---

1. On your form, click the component for which you want to write an event handler.
2. To create the default event for the component, double-click the component on the form.  
  
To choose another event for the component, click the *Events* tab in the *Object Inspector*, locate the event, and double-click its text box.  
  
The *Code Editor* opens.
3. Type the code that will execute when the event occurs at runtime.



# Building an ASP.NET Database Application

---

The following procedure describes the minimum number of steps required to build a simple ASP.NET database application using BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of ASP.NET, ADO.NET, and BDP.NET at work.

Building an ASP.NET application with BDP.NET components consists of four major steps:

1. Create an ASP.NET project.
2. Configure BDP.NET connection components and a data source.
3. Add a DataBind call.
4. Connect a DataGrid to the connection components.

**Tip:** For testing purposes, use the employee.gdb database included with Interbase, if included with your version of the product.

## To create an ASP.NET project

---

1. Choose File ► New ► ASP.NET Web Application.

The *New ASP.NET Application* dialog appears.

2. In the *Name* field, enter the name of your project.
3. In the *Location* field, enter the project path.

**Tip:** Most ASP.NET projects reside in the IIS directory: Inetpub\wwwroot.

## To change Web server settings (optional)

---

1. In the *New ASP.NET Application* dialog, click *View Server Options*

The dialog expands to show additional server options.

2. Set the various read and write attributes of the project as needed or accept the defaults.

**Tip:** In most cases, the default settings will suffice.

3. Click *OK*.

The Web Forms designer appears.

## To configure data components

---

1. Drag and drop a *BdpDataAdapter* component onto the designer. If necessary, select *BdpDataAdapter*.
2. In *Object Inspector*, select *Configure Data Adapter*.  
The *Data Adapter Configuration* dialog appears.
3. If necessary, select the *Command* tab. From the *Connection* drop-down, select *New Connection*.
4. The *Borland Data Provider: Connections Editor* dialog appears.

**Tip:** Alternatively, use Data Explorer to drag and drop a table on to the designer surface. Data Explorer sets the connection string automatically.

## To set up a connection

---

1. In *Borland Data Provider: Connections Editor*, select the appropriate item from the *Connections* list.
2. In *Connection Settings*, enter the *Database* path.

**Note:** If referring to a database on the local disk, prepend the path with *localhost:*. If using Interbase, for example, you would enter the path to your Interbase database: *localhost:C:\Program Files\Borland\Interbase\Examples\employee.gdb* (or whatever the actual path might be for your system).

3. Complete the *UserName* and *Password* fields for the database as needed.
4. Click *Test* to confirm the connection.  
A dialog appears confirming the status of the connection.
5. Click *OK* to return to the *Borland Data Provider: Connections Editor* dialog.
6. Click *OK* to return to the *Data Adapter Configuration* dialog.

In the *Command* tab, the areas for *Tables* and *Columns* are updated with information from your connection.

## To set a command

---

1. In the *Select* area, enter an SQL command.

**Tip:** For Interbase's employee.gdb database, you might enter `select * from SALES`, as an example.

2. Click the *Preview Data* tab.

3. Click *Refresh*.

Column and row data appear.

4. Click the *DataSet* tab.

5. Confirm that *New DataSet* is selected.

6. Click *OK*.

New components for DataSet and BdpConnection appear on the designer.

7. Select BdpDataAdapter component.

8. In *Object Inspector*, select the Active property drop-down and set the value to *True*.

## To connect a DataGrid to a DataSet

---

1. Drag and drop a DataGrid web control onto the designer. If necessary, select DataGrid.

2. In *Object Inspector*, select the DataSource property drop-down. Select the DataSet component that you generated previously (the default is DataSet1).

3. In *Object Inspector*, select the DataMember property drop-down. Select the appropriate table.

The DataGrid displays data from the DataSet.

## To add a DataBind call

---

1. Use the *Object Inspector* drop-down to select the WebForm (*WebForm1* is the default).

2. In *Object Inspector*, select the *Events* tab.

3. Set the Load event to *Page\_Load*.

4. In *Object Inspector*, double-click *Page\_Load*.

The code-behind designer appears, cursor in place between event handler brackets.

5. Code the DataBind call:

```
Self.dataGrid1.DataBind();
```

**Note:** If you are using data aware controls, for instance from a third-party provider, you may not need to code the DataBind call.

6. Choose Run ► Run.

The application compiles and the HTTP server displays a Web Form with the datagrid.

While presenting a minimum number of steps required to build a database project, the preceding procedure demonstrates the major components of the ASP.NET, ADO.NET, and BDP.NET architectures at work, including: providers, datasets, and adapters. The adapter connects to the physical data source via a provider, sending a command that will read data from the data source and populate a dataset. Once populated, a datagrid displays data from the dataset.

Once created, use other BDP.NET designers to modify and maintain the components of your project.

# Building an ASP.NET "Hello world" Application

---

Though simple, the ASP.NET "Hello world" application demonstrates the essential steps for creating an ASP.NET application. The application uses a Web Form, controls, and an event that will display a result in response to a user action.

Building the ASP.NET "Hello world" application consists of five major steps:

1. Create the ASP.NET project.
2. Accept the default Web server settings.
3. Create the ASP.NET page and add components.
4. Create the application logic.
5. Run the application.

## To create an ASP.NET project

---

1. Choose File ► New ► ASP.NET Web Application.

The *New Items* dialog appears.

2. In the *New Items* dialog, select *ASP.NET Web Application*.

3. Click *OK*.

The *New ASP.NET Application* dialog appears.

4. In the *Name* field, enter HelloWorld for the application name.

5. In the *Location* field, accept the default or enter [Inetpub]\HelloWorld, where [Inetpub] is the directory location for IIS projects (for example, C:\Inetpub\wwwroot\HelloWorld).

## To change Web server settings (optional)

---

1. In the *New ASP.NET Application* dialog, click *View Server Options*

The dialog expands to show additional server options.

2. Set the various read and write attributes of the project as needed or accept the defaults.

**Tip:** For most ASP.NET projects, the default settings will suffice.

3. Click *OK*.

The Web Forms designer appears.

## To create the ASP.NET page

---

1. If necessary, select *Design* view. In the *Web Controls* Tool Palette, drag and drop a Button component onto the designer surface.

The Button control appears on the designer surface

2. If necessary, select the Button control. In *Object Inspector*, set the button's *Text* property to Hello, world!.

## To associate code with the button control

---

1. In the designer, double-click the Button control.

The code-behind designer appears, cursor in place between event handler brackets.

2. Code the application logic:

```
button1.Text := button1.Text + ' Hello, developer!';
```

3. Choose File ► Save to save the application.

## To run the "Hello world" application

---

1. Choose Run ► Run.

The application compiles and the HTTP server displays a Web Form in your default browser with the "Hello, world!" button.

2. Click the "Hello, world!" button.

The server updates the page with the response, "Hello, developer!".

3. Close the Web browser to return to the IDE.



# Building an ASP.NET Application

---

The following procedure describes the generic steps required to build a simple ASP.NET project. For more advanced topics, refer to related information following the procedure.

Building an ASP.NET application consists of five major steps:

1. Create an ASP.NET project.
2. Change Web server settings (optional).
3. Create an ASP.NET page.
4. Create the application logic.
5. Run the application.

## To create an ASP.NET project

---

1. Choose File ► New ► ASP.NET Web Application.

The *New Items* dialog appears.

2. In the *New Items* dialog, select *ASP.NET Web Application*.
3. Click *OK*.

The *New ASP.NET Application* dialog appears.

4. In the *Name* field, enter the name of your project.
5. In the *Location* field, enter the project path.

**Tip:** Most ASP.NET projects reside in the IIS directory: Inetpub\wwwroot.

## To change Web server settings (optional)

---

1. In the *New ASP.NET Application* dialog, click *View Server Options*

The dialog expands to show additional server options.

2. Set the various read and write attributes of the project as needed or accept the defaults.

**Tip:** In most cases, the default settings will suffice.

3. Click *OK*.

The Web Forms designer appears.

## **To create an ASP.NET page**

---

1. If necessary, select *Design* view.
2. From the *Tool Palette*, drag and drop components onto the designer to define the user interface.
3. Add code behind logic to components.

## **To add code behind logic to a component**

---

1. In *Design* view, double-click the component to which you wish to apply logic.  
The code-behind designer appears, cursor in place between event handler brackets.

2. Add your logic.

3. Run the application.

The application saves and compiles. Once you compile the application, the generated aspx file displays HTML in the default web browser.

# Building an Application with DB Web Controls

---

The following procedure describes the minimum number of steps required to build a simple ASP.NET database application using DB Web Controls and BDP.NET. After generating the required connection objects, the project displays data in a DBWebGrid with a DBWebNavigator. Additional information is provided for other common DB Web Controls.

Users should already be familiar with creating an ASP.NET project using BDP.NET.

Building the simple ASP.NET application with DB Web Controls and BDP.NET consists of three major steps:

1. Prepare an ASP.NET project with BDP.NET or other connection components.
2. Drag and drop a DBWebDataSource onto the designer and set its DataSource property to a DataSet, DataView or DataTable.
3. Drag and drop a DBWebGrid and other DB Web Control onto the designer.

**Tip:** Dragging and dropping web components places them in absolute position. Double clicking leaves them in flow layout. Flow layout is much easier to manage. For instance, if the controls are at an absolute position and they change sizes at runtime (as might happen to a DataGrid when you add and remove rows), the grid may overwrite other controls.

## To prepare an ASP.NET project for DB Web Controls

---

1. Create an ASP.NET project.
2. Set up BDP.NET or other data access components, setting the DataSource property to an existing DataSet, DataView, or DataTable.

**Tip:** For more information about setting up BDP.NET data access components, see the related procedure for Building an ASP.NET Database Application. Instead of using a DataGrid and adding a [DataBind](#) call, in the following procedure you'll use DB Web Controls without a [DataBind](#) call.


## To configure a DBWebDataSource

---

1. Drag and drop a DBWebDataSource onto the designer.
2. In *Object Inspector*, select the DataSource property. Select an existing data source (for example, DataSet1).

## To configure DB Web Controls

---

1. Drag and drop a DBWebNavigator onto the designer.
2. In *Object Inspector*, select the DBDataSource property drop-down. Select DBWebDataSource1.
3. In *Object Inspector*, select the TableName property drop-down. Select the appropriate Table (for example, Table1).  
**Tip:** If no TableName is available, verify that the BdpDataAdapter Active property is set to true.
4. Drag and drop a DBWebGrid onto the designer.
5. In *Object Inspector*, select the DBDataSource property drop-down. Select the DBWebDataSource (for example, DBWebDataSource1).
6. In *Object Inspector*, select the TableName property drop-down. Select the appropriate Table (for example, Table1).  
The grid displays data.
7. Drag and drop other DB Web Controls as required.
8. Select DB Web Control `DBDataSource`, `TableName`, and other properties as appropriate.  
For data-aware Column Controls (such as DBWebTextBox, DBWebImage, DBWebMemo, and DBWebCalendar) additionally set the `ColumnName` property.  
For data-aware Lookup Column Controls (such as DBWebDropDownList, DBWebListBox, and DBWebRadioButtonList), additionally set the `LookupTableName`, the `DataTextField`, and the `DataValueField`.
9. Choose Run  Run.  
The application compiles and the HTTP server displays a Web Form with DBWebGrid displaying data.

# Extending DB Web Controls

---

The following procedure describes the steps required to extend DB Web Controls. The system for building data-aware web controls is designed to be extensible. To build a new data-aware web control requires only a single class, that which implements the new control.

Extending DB Web Controls consists of four major steps:

1. Create a new class.
2. Update the `using` section.
3. Specify the inheritance schema of the class.
4. Override and implement appropriate members.

## To create a new class

---

1. Choose File ► New ► Class.

The *New Class* dialog appears.

2. Specify the correct name space.

3. Save the file.

The *New Class* designer appears.

## To update the the using section:

---

1. Click *Code*.

The *Code Editor* appears.

2. Add the following to the `using` section:

```
using System;
using System.Data;
using System.IO;
using System.Collections;
using System.Collections.Specialized;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using Borland.Data.Web; // add reference to this to project
using System.ComponentModel;
using System.ComponentModel.Design;
```

```
using System.Drawing;
using System.Drawing.Design;
```

## To specify the inheritance schema of the class

---

1. In the constructor, create an instance of the IDBWebDataLink (or IDBWebColumnLink) interface.

**Note:** You can use the existing DBWebControlDesigner as it relies only on the control having implemented one of the link interfaces.

2. Use the following code:

```
[Designer("Borland.Data.Web.DBWebControlDesigner"),
ToolboxData("<{0}:MyControl runat=server></{0}:MyControl>")]

public class MyControl:ListBox, IDBColumnLink, IPostBackDataHandler
{
    private DBWebColumnLink FColumnLink;
        private IDBWebColumnLink IColumnLink;

    public MyControl() : base()
    {
        FColumnLink = new DBWebColumnLink(this);
        IColumnLink = (FColumnLink as IDBWebColumnLink);
    }

}
```

3. Override [OnInit](#) for handling postbacks.

```
protected override void OnInit(EventArgs e)
{
    base.OnInit(e);
    if( Page != null )
        Page.RegisterRequiresPostBack(this);
}
```

4. Override [OnPreRender](#) so it calls [DataBind\(\)](#) before the control is rendered. Also, if the control is going to update a column value, then [RegisterHiddenField](#) needs to be called.

```
protected override void OnPreRender(EventArgs args)
{
    base.OnPreRender(args);
    Page.RegisterHiddenField(this.ID, IColumnLink.TableName + DBWebConst.
```

```

        Splitter + IColumnLink.ColumnName);
        DataBind();
    }

```

5. Implement `IPostBackHandler` and in the `LoadPostData` event store the `postCollection` values for future use by the `DBDataSource`.

```

// RaisePostDataChangedEvent is called prior to DataBind()
// DataSet and related properties are NOT available here
// Child controls are not available
bool IPostBackDataHandler.LoadPostData(string postDataKey,
    NameValueCollection postCollection)
{
    FColumnLink.LoadPostData(postDataKey, postCollection);
    return false;
}

void IPostBackDataHandler.RaisePostDataChangedEvent()
{
}

```

6. Implement `IDBWebDataLink`, `IDBWebColumnLink`, and so-on, and provide public access methods to their properties:

**Note:** Specify the use of the `Borland.Data.Web.TableNamePropEditor` and `Borland.Data.Web.ColumnNamePropEditor` where appropriate.

```

string IDBWebDataLink.TableName
{
    get
    {
        return IDataLink.TableName;
    }
    set
    {
        IDataLink.TableName = value;
    }
}
IDBDataSource IDBWebDataLink.DBDataSource
{
    get
    {
        return IDataLink.DBDataSource;
    }
    set
    {

```

```

        IDataLink.DBDataSource = value;
    }
}
[Editor(typeof(Borland.Data.Web.TableNamePropEditor), typeof(
UITypeEditor)), Category("DBWebControl"),
DefaultValue(null)]
public string TableName
{
    get
    {
        return IDataLink.TableName;
    }
    set
    {
        IDataLink.TableName = value;
    }
}
[Category("DBWebControl"),
DesignerSerializationVisibility(DesignerSerializationVisibility.
Hidden), DefaultValue(null)]
public IDBDataSource DBDataSource
{
    get
    {
        return IDataLink.DBDataSource;
    }
    set
    {
        IDataLink.DBDataSource = value;
    }
}
}

```

## 7. Implement `DataBind()`.

```

public override void DataBind()
{
    // must be done first so DBDataSource setter is triggered.
    base.DataBind();
    if( FColumnLink.IsDataBound )
    {
        // Figure what data you want to be put into the control.
    }
}

```



8. Optionally, override the `Render` method. If all you only need to set the data from `DBDataSource` settings, then you don't need to override `Render`. If you want to code special behavior, then you can override `Render` and write the output anyway you wish. Following is a sample used for the `DBWebGrid` so that a special grid is generated at design time when the properties for the `IDBWebDataLink` interface have not yet been set:

```
protected override void Render(HtmlTextWriter output)
{
    if( ClassUtils.IsDesignTime(Page) && !FDataLink.IsDataBound )
    {
        FDataGrid.EditItemIndex = 0;
        FDataGrid.DataBind();
        FDataGrid.RenderControl(output);
    }
    else
        base.Render(output);
}
```

The DB Web Control is now extended.

# Using the HTML Tag Editor

---

In addition to the aspx code editor that is included every time you create a new ASP.NET Web Forms application, Delphi 8 for .NET provides an HTML Tag Editor, which is a re-sizable window at the bottom of the Web Forms Designer. The tag editor allows you to review and modify HTML tags at the same time you view your controls, rather than forcing you to switch back and forth between the aspx code editor and the Web Forms Designer.

## To use the HTML Tag Editor

---

1. View HTML code for an individual control.
2. View HTML code for all controls.
3. Modify a control.
4. Change editor properties.
5. Zoom between contents in the form and the form container.

## To view HTML code for an individual control

---

1. With a new ASP.NET application open, click, drag, and drop an HTML element from the Tool Palette to the Web Form Designer surface.

The editor displays the HTML code, highlighted in a red font, for this control in its gray header bar.

2. To view the individual control's code, click anywhere on the Web Form Designer surface to deselect the control.

The HTML code appears in the tag editor window, with syntax highlighting. The gray header of the tag editor now displays the higher level tag, usually the FORM tag that defines this particular Web Form.

**Note:** If a control is defined using several lines of HTML code, when you select the control, the first line of the code is displayed in the gray header of the tag editor. The additional code appears below in the tag editor window.

## To view the HTML code for all controls

---

1. With a new ASP.NET application open, click, drag, and drop several HTML elements from the Tool Palette to the Web Form Designer surface.

The editor displays the HTML code for each element, highlighted in a red font, in its gray header, for each control as you drop them on the Web Form Designer surface.

2. Click anywhere on the Web Form Designer surface to deselect all controls.

This displays the code for all the controls in the tag editor, with syntax highlighting.

## To modify a control

---

1. Click anywhere on the Web Form Designer surface to deselect all controls.
2. Locate the tag that corresponds to the control you want to modify.
3. Modify the code, and the change is immediately reflected in the control on the Web Form Designer surface.
4. Save your project to make the modifications permanent.

## To change editor properties

---

1. With the insertion pointer anywhere in the tag editor window, right-click to display the context-sensitive pop-up menu.
2. Select Properties.
3. Change any code editor properties.
4. Click *OK*.

Changes take effect immediately.

## To zoom between contents of the form and the form container

---

1. To zoom out so that you can view the HTML form definition, click the left-hand blue arrow in the gray header of the tag editor.

**Note:** You can only use this feature when the insertion pointer is somewhere in the tag editor, rather than on the Web Form Designer surface.

2. To zoom in so that you can view only the content within the FORM tags, click the right-hand blue arrow in the gray header of the tag editor.

**Note:** You can only use this feature when the insertion pointer is somewhere in the tag editor, rather than on the Web Form Designer surface.

# Compiling and Building Applications

---

# Adding Languages to a Project

---

You can add languages to your project by using the Satellite Assembly Wizard. For each language that you add, the Satellite Assembly Wizard generates a separate satellite assembly project in your project group. Each satellite assembly project is given an extension based on the language's locale.

## To add a language to a project

---

1. Save and build your project.
2. With your project open in the IDE, choose Project ► Languages ► Add. Alternatively, you can choose File ► New ► Other ► Current Project ► Satellite Assembly Wizard.  
The Satellite Assembly Wizard is displayed.
3. Make sure your project is selected in the list that appears in the dialog and then click *Next*.
4. Click the checkbox next to the languages that you want to add to your project and then click *Next*.
5. Review the directory path information that the wizard will use for the language's satellite assembly.  
**Tip:** To change the path, click the path, and then click the ellipsis button to browse to a different directory.  
When you are satisfied with the path information, click *Next*.
6. If no satellite assembly for the language exists yet, *Create New* appears in the *Update Mode* column. Click *Next*.  
If a satellite assembly exists for the language in the directory you have specified, click in the *Update Mode* column to select *Update* or *Overwrite*. Choose *Update* to keep and modify the existing satellite assembly project. Choose *Overwrite* to create a new, empty project and to delete the old project and any translations it contains. Click *Next*.
7. Review the summary of what the wizard will do and click *Finish* to create or update the satellite assemblies for the languages you have selected.  
If the wizard asks to generate a drcil file, click *Yes*. Any project that uses its own resource strings (instead of previously compiled rc files) needs a drcil file.  
If you are sure that no new drcil files are needed (because your project does not introduce any resource strings of its own), select *Skip drcil files that are not found* in the final dialog. This prevents the wizard from generating, or asking to generate, drcil files.
8. Click *Yes* to compile. Click *Yes* again to save your project group.

The generated projects contain untranslated copies of the resource strings in your original project. By default, the Translation Manager is displayed, enabling you to begin translating the resource files.

## To remove a language from a project

---

1. Open your project.
2. Select Project ► Languages ► Remove.
3. Check the languages that you want to remove and then click *Next*.
4. Click *Finish*.

The Satellite Assembly Wizard removes the selected satellite assemblies from your project file, but does not delete the assemblies, the source of the assemblies, or the directories in which they reside.

## To restore a language to a project

---

1. Choose Project ► Languages ► Add to start the Satellite Assembly Wizard.
2. Specify the directory path of the old satellite assembly in the appropriate dialog.
3. In the *Update Mode* column, select *Update*.

If a satellite assembly already exists for the language (in the directory you have specified), click in the *Update Mode* column to select *Update* or *Overwrite*. Choose *Update* to keep and modify the existing assembly project. Choose *Overwrite* to create a new, empty project and to delete the old project and any translations it contains.

# Building Packages

---

You can create packages easily in Delphi 8 for .NET and include them in your projects.

## To create a new package

---

1. Choose File ► New ► Other ► Delphi for .NET Projects.

2. Double-click the Package icon in the Gallery.

This creates a new, empty package and makes an entry for it in the *Project Manager*, along with two folders: one marked Contains and one marked Requires.

**Note:** If you want to add required files to the package, you must add compiled packages (.dcpil, .dll) to the *Required* folder. Add uncompiled code files (.pas) to the *Contains* folder.

3. Select the package name in the Project Manager.
4. Right-click to display the drop down context menu.
5. Select *Add....*

This displays the *Add Package* dialog.

6. Browse to locate the file or files you want to add.
7. Select one or more files, and click *Open*.
8. Click *OK*.

This adds the selected files to the package.

9. Choose Project ► Build <Package Name> to build the package.

## To add a package to a project

---

1. Choose File ► New ► Other ► VCL Forms Application.
2. Select the project name in the *Project Manager*.
3. Right-click to display the drop down context menu.
4. Choose *Add....*



5. Browse to locate a package file.
6. Select the file and click *Open*.
7. Click *OK*.  
This adds the package to the project.
8. Choose Project ► Build <Project Name> to build the project.

## To add a component package to the Tool Palette

---

1. Choose Components ► Installed .NET Components.
2. Click the *.NET VCL Components* tab.
3. Click *Add....*
4. Locate the package file you want to add to the *Tool Palette*.
5. Click *Open*.  
This displays the available components from the package.
6. Click *OK*.  
The components appear in the *Tool Palette*.

# Editing Resource Files in the Translation Manager

---

After you have added languages to your project by using the Satellite Assembly Wizard, you can use the Translation Manager to view and edit your resource files. Within the Translation Manager, you can edit resource strings directly, add translated strings to the Translation Repository, or get strings from the Translation Repository.

## To edit resource strings

---

1. Open a project that includes languages that were added by using the Satellite Assembly Wizard.
2. Choose View ► Translation Manager.
3. In the Translation Manager, click the *Workspace* tab.
4. Expand the project tree view to display the resource files that you want to edit. The resource files are listed under the *.NET Resources* node.  
**Tip:** Use the expand and collapse icons on the toolbar above the tree view.
5. Click the resource file you want to edit. The resource strings in the file are displayed in a grid in the right pane.
6. Click the field that you want to edit and do one of the following:
  - type the new text directly in the grid
  - right-click the field and choose Edit to edit the string in a dialog box
  - click the **Multi-line editor** icon on the toolbar above the grid
7. Optionally, enter a comment in the *Comment* field.
8. Optionally, set the translation status for the string by using the drop-down list in the *Status* field.
9. Click the *Save Translation* icon on the toolbar above the grid to update the resource file.

**Tip:** To display the original form or translated form, click the *Show original form* and *Show translated form* icons in the toolbar above the grid.

## To add a resource string to the Translation Repository

---

1. After editing a resource string in the Translation Manager (see above), right-click the string that you want to add to the Translation Repository.

2. Choose Repository ► Add strings to repository.

The resource string is added to the Translation Repository and can be viewed by closing the Translation Manager and choosing Tools ► Translation Repository.

## To get a resource string from the Translation Repository

---

1. In the Translation Manager, click the *Workspace* tab.
2. Expand the project tree view to display the resource files that you want to edit. The .resx files are listed under the *.NET Resources* node. The .nfm files are listed under the *Forms* node.
3. Click the resource file you want to edit. The resource strings in the file are displayed in a grid in the right pane.
4. Right-click the field that you want to update and choose Repository ► Get strings from repository.

If the Translation Repository contains only one translation that matches the selected source string, it copies that translation into the target language column. If the Repository contains more than one match for the selected resource, its default behavior is to retrieve the first matching translation it finds.

**Tip:** To change this behavior, close the Transaction Manager and choose Tools ► Translation Tools Options, click the *Repository* tab, and change the *Multiple Find Action* setting.

## To open the resource file in a text editor

---

1. In the Translation Manager, click the *Project* tab.
2. Click the *Files* tab.
3. Double-click the resource file that you want to update. The file opens in a text editor.
4. Change the file as needed and save it.

**Tip:** To change the text editor used by the Translation Manager, choose Tools ► Translation Tools Options and change executable file specified in the *External Editor* field.

# Setting Up the External Translation Manager

---

If you are localizing an application and do not have the Delphi 8 for .NET IDE, you can use the External Translation Manager (ETM) to perform the translations. The ETM provides the same basic functionality as the Translation Manager in the product. To use the ETM, the developer must provide you with the required ETM and project files. After you have finished the translations, you can send the translated files back to the developer to add to the project.

The basic steps to set up the ETM are:

1. Obtain from the developer copies of the files listed in the table below.
2. Copy files onto your machine.
3. Run the ETM.

---

**From C:\Program Files\Borland\BDS\2.0\Bin:**

---

Borland.Delphi.dll

---

Borland.Globalization.dll

---

Borland.ITE.dll

---

Borland.ITE.FormDesigner.dll

---

Borland.Vcl.dll

---

Borland.VclRtl.dll

---

designide71.bpl

---

dfm71.bpl

---

DotnetCoreAssemblies71.bpl

---

itecore71.bpl

---

itedotnet71.bpl

---

ResX71.bpl

---

rtl71.bpl

---

vcl71.bpl

---

vclactnband71.bpl

---

vclide71.bpl

---

vcx71.bpl

---

xmlrtl71.bpl

---

etm71.ex

---

---

\*71.bpl

---

**A translation kit or zipped file containing the following project files:**

---

satellite assembly created using the Satellite Assembly Wizard for each language to be translated

---

.bdsproj project file generated using File ► Save as in the ETM project

---

standalone translation repository (\*.tmx) files

---

**Note:** The developer saves each satellite assembly in a separate root directory below the project directory. When you return the translated files to the developer, the developer copies the translated project into each language's root directory.

## To copy the files to your machine

---

1. Create a directory, such as C:\ETM, and copy the files from the Bin and System directories into it.
2. Unzip the translation kit into a directory, such as C:\ETM.

## To run the ETM from the command line or from Windows Explorer

---

1. To run the ETM from the command line, enter: `etm71.exe [files]`  
For the files parameter, you can enter the project group files or the project files.
2. To run the ETM from Windows Explorer, double-click `etm71.exe`

# Database

---

# Adding a New Connection to the Data Explorer

---

You can add new connections to the *Data Explorer*, which persist as long as the connection object exists.

## To add a new connection

---

1. Choose View ► Data Explorer.

This displays the *Data Explorer*.

2. Select a provider from the tree list.
3. Right-click to display a pop-up menu.
4. Choose *Add New Connection*.

This displays the *Add New Connection* dialog.

5. Enter the name of the new connection.
6. Click *OK*.

**Tip:** If you need to modify your new connection settings, right-click on your new connection and scroll down to *modify a connection*. A *Connection Editor* dialog appears. Enter your connection settings and click *OK*.

# Browsing a Database in the Data Explorer

---

Once you have a live connection, you can use the *Data Explorer* to browse database objects.

## To browse database objects

---

1. Choose View ► Data Explorer.
2. Double-click a provider name to expand the list of available connections.
3. Double-click a connection name to expand the list of database objects.

**Note:** If you receive an error because your connection is not live, you should refresh your provider.

## To retrieve data from the database

---

1. Expand a connection in the *Data Explorer*.
2. Double-click a table name or view name to retrieve data.

This operation returns a result set into a tabbed *Data Explorer* page in the *Code Editor*.

**Tip:** You can also select a table in the *Data Explorer* and right-click to display a pop-up menu with a *Retrieve Data From Table* command.

## To run a stored procedure

---

1. Choose View ► Data Explorer.
2. Expand a connection in the *Data Explorer* and locate a stored procedure.

The expanded procedure displays a list of the parameters.

3. Double-click the procedure name to execute the procedure.

The result set appears in a *SQL Window* in the *Code Editor*.

**Tip:** You can also select a procedure in the *Data Explorer* and right-click to display a pop-up menu with an *Execute* command.



# Building a Windows Forms Database Application

---

The following procedure describes the minimum number of steps required to build a simple ADO.NET application using Windows Forms and BDP.NET. After generating the required connection objects, the project displays data in a DataGrid.

BDP.NET includes component designers to facilitate the creation of database applications. Instead of dropping individual components on a designer, configuring each in turn, use BDP.NET designers to rapidly create and configure database components. The following procedure demonstrates the major components of Windows Forms, ADO.NET, and BDP.NET at work.

Building a BDP.NET project consists of two major steps:

1. Configure BDP.NET connection components and a data source.
2. Connect a DataGrid to the connection components.

**Tip:** For testing purposes, use the `employee.gdb` database included with Interbase, if included with your version of the product.

## To configure connection components and a data source

---

1. Choose File ► New ► Windows Forms Application.

The Windows Forms designer appears.

2. Drag and drop a `BdpDataAdapter` component onto the designer. If necessary, select `BdpDataAdapter`.

3. In *Object Inspector*, select *Configure Data Adapter*.

The *Data Adapter Configuration* dialog appears.

4. If necessary, select the *Command* tab. From the *Connection* drop-down, select *New Connection*.

5. The *Borland Data Provider: Connections Editor* dialog appears.

**Tip:** Alternatively, use Data Explorer to drag and drop a table on to the designer surface. Data Explorer sets the connection string automatically.

## To set up a connection

---

1. In *Borland Data Provider: Connections Editor*, select the appropriate item from the *Connections* list.
2. In *Connection Settings*, enter the *Database* path.

**Tip:** If using Interbase, you would enter the path to your Interbase database.

3. Complete the *UserName* and *Password* fields for the database as needed.
4. Click *Test* to confirm the connection.

A dialog appears indicating the status of the connection.

5. Click *OK* to return to the *Data Adapter Configuration* dialog.

In the *Command* tab, the areas for *Tables* and *Columns* are updated with information from your connection.

## To set a command

---

1. In the *Select* area, enter an SQL command.

**Tip:** For Interbase's employee.gdb database, you might enter `select * from SALES`, as an example.

2. Click the *Preview Data* tab.

3. Click *Refresh*.

Column and row data appear.

4. Click the *DataSet* tab.

5. Confirm that *New DataSet* is selected.

6. Click *OK*.

New components for *DataSet* and *BdpConnection* appear on the designer.

7. Select *BdpDataAdapter* component.

8. In *Object Inspector*, select the *Active* property drop-down and set the value to *True*.

## To connect a DataGrid to a DataSet

---

1. Drag and drop a *DataGrid* component onto the designer. If necessary, select *DataGrid*.

2. In *Object Inspector*, select the DataSource property drop-down. Select the DataSet component that you generated previously (the default is DataSet1).
3. In *Object Inspector*, select the DataMember property drop-down. Select the appropriate table.  
The DataGrid displays data from the DataSet.
4. Select Run ► Run.  
The application compiles and displays a Windows Form with DataGrid.

While presenting a minimum number of steps required to build an ADO.NET project, the preceding procedure demonstrates the major components of the Windows Forms, ADO.NET, and BDP.NET architectures at work, including: providers, datasets, and adapters. The adapter connects to the physical data source via a provider, sending a command that will read data from the data source and populate a dataset. Once populated, a datagrid displays data from the dataset.

Once created, use other BDP.NET designers to modify and maintain the components of your project.

# Creating Database Projects from the Data Explorer

---

You can drag and drop data from the *Data Explorer* to any forms such as Windows Forms or Web Forms, and Global.asax files, to populate datasets and quickly build a database project. This allows you to automatically hook up database components to your project and eliminates the need to provide a connection string, which can be prone to errors if entered manually.

## To create a database project from the Data Explorer

---

1. Make sure you have a live connection to a database.
2. From the View menu, select *Data Explorer*.
3. Choose File ► New ► Other and select a Delphi for .NET project.
4. Expand the *Data Explorer Tree* by drilling down to the *Table* or *View* level.
5. Drag and drop any data onto your form.

All the appropriate database components, including `bdpConnection`, `bdpCommand`, and `bdpDataAdapter`, appear in the component tray.

6. Specify the appropriate database properties for each database component, for instance, set the *Active* property to *True*.

**Note:** A `DataGrid` will not appear automatically so make sure you drop a `DataGrid` component onto your form to appropriately display data, when necessary.

# Creating Table Mappings

---

Using the TableMappings property, you can map columns between a data source and an in-memory dataset. This allows you to use different, often more descriptive names for your dataset columns. You can also map a column in a database table to a column in the dataset different from that which is selected by default. The TableMappings property also allows you to create a dataset that contains fewer or more columns than those retrieved from the database schema.

## To create a table mapping

---

1. Create an application.
2. Add and configure database components.
3. Set the table mappings in the TableMappings dialog.

**Note:** This procedure assumes you are using BDP.NET database components.

## To create an application

---

1. Choose File ► New ► Windows Forms Application.
2. Click the Data Explorer tab to display your data sources.
3. Expand the list and locate a live data source.
4. Drag-and-drop a table name onto your Windows Form to add a data source to your application.

You should see two objects in the *Component Tray*: a BdpDataAdapter and a BdpConnection.

For more information about how to create database applications, refer to the additional ADO.NET and database topics in this Help system.

## To configure the database components

---

1. Select the BdpDataAdapter icon in the *Component Tray*.
2. Click the *Configure Data Adapter...* designer verb to open the Data Adapter Configuration dialog.
3. Select the *DataSet* tab.
4. Click the *New DataSet* radio button.

5. Click *OK*.

This creates a new dataset and displays an icon for it in the *Component Tray*.

## To set table mappings

---

1. Select the *BdpDataAdapter* icon in the *Component Tray*.
2. Double-click the *Collections* field for the *TableMappings* property in the *Object Inspector*.

This displays the *TableMappings* dialog.

3. If you want to use an existing dataset as a model for the columns, check the *Use a dataset to suggest table and column names* checkbox.

This provides you with a list of column names from an existing dataset based on the schema of that dataset. The column names are not linked to anything when you use this process.

4. If you checked the *Use a dataset to suggest table and column names* checkbox, you can choose the dataset from the *DataSet* drop down list.

5. Select the source table from the *Source table* drop down list.

If there is more than one table in the data source, their names appear in the drop down list.

6. If you chose to use a dataset to suggest table and column names, and that dataset contains more than one table, you can select the table you want to use from the *Dataset table* drop down list.

The column names from the source table and from the dataset should appear in the *Column mappings* grid. As they are displayed by default, they represent the mapping from source to dataset; in other words, the data adapter reads data from each column named on the left side of the grid and stores the data in the dataset column named in the corresponding field on the right side of the grid. You can change the names on either side by typing new names or by selecting different tables. This allows you to store queried data into different dataset columns than the ones created in the dataset by default.

7. If you want to modify a mapping, type a new name in the *Dataset table* column next to the target *Source table* column.

This results in the data from the *Source table* column being stored in the new dataset column.

**Note:** If you want to reset the column names so that the dataset columns match the data source columns, you can click the *Reset* button.

## To delete a mapping

---

1. Select the grid row that you want to delete.
2. Click *Delete*.

This will cause the query to ignore that column in the source table and to not fill the dataset column with any data.

# Executing SQL in the Data Explorer

---

You can write, edit, and execute SQL in an *SQL Window*, which is available from within the *Data Explorer*.

## To open a SQL Window

---

1. Choose View ► Data Explorer.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose *SQL Window*.

This opens a tabbed *SQL Window* in the *Code Editor*.

## To execute SQL

---

1. Enter a valid SQL statement or stored procedure name in the multi-line text box at the top of the *SQL Window*.
2. Click *Execute*.

If the SQL statement or stored procedure is valid, the result set appears in the bottom pane of the *SQL Window*.

**Note:** The SQL statement or stored procedure must operate against the current connection and its target database. You cannot execute SQL against a database to which you are not connected.

3. Click *Clear* to clear the SQL statement or stored procedure from the multi-line text box.



# Handling Errors in Table Mapping

---

Whenever you perform any type of comparison function between a data source and an in-memory data representation, there is potential for error. Errors can occur when a data source and its corresponding dataset do not share uniform numbers of columns, or when column types in a data source do not correspond to the column types in the dataset. In addition, other, internal errors can occur for which there is no design-time workaround. You can use both the `MissingMappingAction` property and the `MissingSchemaAction` property to respond to errors in your table mapping operations. Use the `MissingMappingAction` when you want to specify how the adapter should respond when the mapping is missing. Use the `MissingSchemaAction` when you want to specify how the adapter should respond when it tries to write data to a column that isn't defined in the dataset.

## To set the `MissingMappingAction` property

---

1. Once you have created a `BdpDataAdapter` and have set up your table mappings, click the drop down list next to the `MissingMappingAction` property in the Object Inspector.
2. Select *Passthrough* if you want the adapter to load the data source column data into a dataset column of the same name, or, if there is no corresponding dataset column, if you want the adapter to perform the action specified in the `MissingSchemaAction` property.
3. Select *Ignore* if you want to keep data from being loaded when data source columns are not properly mapped to dataset columns.

This could occur if mapped columns are of incompatible data types, lengths, or have other errors.

4. Select *Error* if you want the adapter to raise an error that you can trap.

## To set the `MissingSchemaAction` property

---

1. Select *Add* if you want the data source table or column added to the dataset and its schema.  
Setting the `MissingMappingAction` property to *Passthrough* and the `MissingSchemaAction` to *Add* results in a duplication of data source table and column names in the dataset.
2. Select *AddWithKey* if you want the data source table or column added to the dataset and its schema along with the table's or column's primary key information.
3. Select *Ignore* if you don't want a table or column added to the dataset, when that table or column aren't already represented in the dataset schema.

Specify *Ignore* when you want the dataset loaded only with data explicitly specified in the table mappings. This may be necessary if your adapter calls a stored procedure or a user-defined SQL statement that returns more columns than are defined in the dataset.

4. Select *Error* if you want the adapter to raise an error that you can trap.

# Modifying Connections in the Data Explorer

---

You can modify connections in a variety of ways from the *Data Explorer*.

## To modify connections

---

1. Choose View ► Data Explorer.
2. Select a provider.
3. Right-click to display a pop-up menu to view your options.

## To refresh a connection

---

1. Choose View ► Data Explorer.
2. Select a provider.
3. Right-click to display a pop-up menu.
4. Choose *Refresh*.

This operation reinitializes all connections defined for the selected provider.

## To delete a connection

---

1. Choose View ► Data Explorer.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose *Delete Connection*.

This displays a confirmation message that asks if you want to delete the connection.

5. Click *OK*.

## To modify a connection

---

1. Choose View ► Data Explorer.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose *Modify Connection*.  
This displays the *Connections Editor* dialog.
5. Make changes to the appropriate values in the editor.
6. Click *OK*.

## To close a connection

---

1. Choose View ► Data Explorer.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose *Close Connection*.  
If the connection is open, this operation closes it.

**Note:** If the *Close Connection* command is disabled in the menu, the connection is not open.

## To rename a connection

---

1. Choose View ► Data Explorer.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose *Rename Connection*.  
This displays *Rename Connection* dialog.

5. Enter a new name.

6. Click *OK*.

The *Data Explorer* displays the connection with its new name.

# Passing Parameters in a Database Application

---

The following procedures describe a simple application that allows you to pass a parameter value at runtime to a DataSet. Parameters allow you to create applications at design time without knowing specifically what data the user will enter at runtime. This example process assumes that you already have your sample Interbase Employee database set up and connected. For purposes of illustration, this example uses the default connector IBConn1, which is set to a standard location. Your database location may differ.

## To pass a parameter

---

1. Create a data adapter and connection to the Interbase employee.gdb database.
2. Add a text box control, a button control, and a data grid control to your form.
3. Configure the data adapter.
4. To add a parameter to the data adapter.
5. Configure the data grid.
6. Add code to the button Click event..
7. Compile and run the application.

## To create a data adapter and connection

---

1. Choose File ► New ► Application where *Application* is the language type you are using.
2. Click on the *Data Explorer* tab and drill down to find the IBConn1 connection under the Interbase node.
3. Drag and drop the EMPLOYEE table onto the Windows Form.  
This creates a BdpDataAdapter and BdpConnection and displays their icons in the *Component Tray*.
4. Select the data adapter icon, then click the *Configure Data Adapter...* designer verb in the *Designer Verb* area at the bottom of the Object Inspector.  
This displays the *Data Adapter Configuration* dialog.
5. Rewrite the SQL statement that is displayed in the Select tab of the dialog to:

```
SELECT EMP_NO, FIRST_NAME, LAST_NAME, SALARY FROM EMPLOYEE WHERE  
FIRST_NAME = ?;
```

As you can see, this statement is limiting the number of fields. It also contains a `?` character as part of the Where clause. The `?` character is a wildcard that represents the parameter value that your application passes in at runtime. There are at least two reasons for using a parameter in this way. The first reason is to make the application capable of retrieving numerous instances of the data in the selected columns, while using a different value to satisfy the condition. The second reason is that you may not know the actual values at design time. You can imagine how limited the application might be if we retrieved only data where `FIRST_NAME = 'Bob'`.

6. Click the *DataSet* tab.
7. Click *New DataSet*.
8. Click *OK*.

This creates the DataSet that represents your query.

## To add a parameter to the data adapter

---

1. Select the data adapter icon, then expand the properties under SelectCommand in the *Fill* area of the *Object Inspector*.  
You should be able to see your Select statement in the SelectCommand property drop down list box.
2. Change the ParameterCount property to 1.
3. Click the (Collection) entry next to the Parameters property.  
This displays the *BdpParameter Collection Editor*.
4. Click *Add* to add a new parameter.
5. Rename the parameter to *emp*.
6. Set BdpType to *String*, DbType to *Object*, Direction to *Input*, Source Column to *FIRST\_NAME*, and ParameterName to *emp*.
7. Click *OK*.
8. In the *Object Inspector*, set the Active property under Live Data to **True**.

## To add controls to the form

---

1. Drag and drop a TextBox control onto the form.
2. Drag and drop a Button onto the form.
3. Change the Text property of the button to *Get Info*.
4. Drag and drop a DataGridView data control onto the form.
5. Arrange the controls how you want them to appear, making sure that the DataGridView is long enough to display four fields of data.

## To configure the data grid

---

1. Select the data grid.
2. Set the DataSource property to the name of the DataSet (dataSet1 by default).
3. Set the DataMember property to *Table1*.

This should display the column names of the columns specified in the SQL statement that you entered into the data adapter.

## To add code to the button Click event

---

1. Double-click the button to open the Code Editor.
2. In the button1\_Click event code block, add the following code:

```
Self.bdpSelectCommand1.Close();
/* This closes the command to make sure that we will pass the parameter
to */
/* the most current
bdpSelectCommand.

*/

Self.BdpDataAdapter1.Active := false;
/* This clears the data adapter so that we don't maintain old
data */
```



```
Self.bdpSelectCommand1.Parameters['emp'].Value := textBox1.  
Text;  
/* This sets the parameter value to whatever value is in the text  
field. */
```

```
Self.BdpDataAdapter1.Active := true;  
/* This re-activates the data adapter so the refreshed data appears in  
the data grid. */
```

If you have changed the names of any of these items, you need to update these commands to reflect the new names.

3. Save your application.

## To compile and run the application

---

1. Press Shift + F9 to compile the application.
2. Press F9 to run the application.
3. Type one of the names John, Robert, Roger, Kim, Terri, Katherine, or Ann into the text box.
4. Click the button.

This displays the employee number, first name, last name, and salary of the employee with that name in the data grid. If there is more than one person with the same first name, the grid displays all occurrences of employees with that name.

# Using the Data Adapter Preview

---

Borland Delphi 8 for .NET provides a tool that enables communication between a data source and a dataset. You can use the *Data Adapter Preview* to specify what data to move into and out of the dataset either in the form of SQL statements or stored procedures that are invoked to read or write a database.

## To use the Data Adapter Preview

---

1. After you have dropped a *BdpDataAdapter* component onto the designer, click the *Configure Data Adapter* designer verb that appears at the bottom of the *Object Inspector*.
2. Click the *Preview* tab to display the *Data Adapter Preview*.
3. To limit the number of rows fetched, click the *Limit rows* check box.
4. Enter the number of rows you want the result set to contain, in the *Rows to fetch* text box.
5. Click *Refresh* to re-execute the query and to refill the list box with the new number of rows.

# Using the CommandText Designer

---

In order to create a DataSet, your BdpDataAdapter needs to have at least a SQL Select statement defined for the CommandText property. This statement, once built, appears as the CommandText property of the BdpCommand object for the BdpDataAdapter. You can enter this Select statement manually, or you can use the *CommandText* designer to construct the statement, along with Update, Insert, and Delete statements, using a simple point-and-click mechanism. Using this method, once you have a connection to a live data source, you will be able to see the names of tables and columns in the *CommandText* designer. You can pick from listboxes to build the statement. Also, if you create your BdpDataAdapter using the *Data Explorer* and a live connection to a data source, a boilerplate Select statement is created for you in the form `select * from tablename`. You can use this statement to return all rows from the named data source, or you can modify the statement prior to generating the DataSet.

## To generate the commands

---

1. Select a connection from the *Connection* drop-down list box. This must be a BdpConnection you have already defined. Your associated BdpDataAdapter object must also be defined and must have the DataSet Active property set to **True**.

This populates the Tables and Columns list boxes with data from the database.

2. Select a table from the Tables list box.
3. Select each column that you want to appear in your SQL statements.  
As you select the column names, they appear in the SQL text box.
4. Select the check box next to each statement type you want to generate.
5. Click the *Generate SQL* button.

# Using the Data Adapter Designer

---

The Data Adapter contains, at a minimum, a SQL Select statement of the SELECT command property. You can enter this statement yourself, or using the *Data Adapter* designer you can construct the Select, along with the Update, Insert, and Delete statements. The BdpCommandBuilder constructs the Update, Insert, and Delete statements based on the tables and columns you have selected. The *Data Adapter* designer uses a live connection to retrieve metadata from which you can build the appropriate SQL statements for manipulating the data you want to move from a DataSet back into your database.

## To invoke the commands

---

1. Select a connection from the *Connection* drop-down list box. This must be a BdpConnection you have already defined.

This populates the Tables and Columns list boxes with data from the database.

2. Select a table from the Tables list box.
3. Select each column that you want to appear in your SQL statements.
4. Select the check box next to each statement type you want to generate.
5. Click the *Generate SQL* button.
6. Edit the generated text if desired, or reselect different columns and click *Generate SQL* again.
7. Click *OK*.

**Note:** Command components are automatically created as needed based on the selections in the dialog.

# Using the Connection Editor Designer

---

Each connection object can support multiple named connections. These connections can represent connections to multiple databases and database types.

## To add a new connection

---

1. Select an existing *BdpConnection* component in the designer, or drop a *BdpConnection* component onto the designer to create a new object.
2. Click the component designer verb at the bottom of the *Object Inspector* to display the *Connection Editor* dialog.
3. Click *Add* to display the *Add New Connection* dialog.
4. Select a provider from the *Provider Name* drop-down list box.
5. Enter a new name for the connection in the *Connection Name* text box.
6. Click *OK*.
7. Enter the appropriate values for your particular data source.
8. Click *OK*.

## To remove a connection

---

1. Select the connection type until it is highlighted.
2. Click *Remove*.  
A *Confirm Delete* dialog box appears.
3. Click *Yes*.

## To rename a connection

---

1. Right-click on the connection and choose *Rename*.
2. Type the new name of the connection.

3. Click OK.

# Using Standard DataSets

---

The standard DataSet provides an in-memory representation of one or more tables or views retrieved from a connected data source. Because of the level of indirection used in coding the underlying data structure, you are only able to see the column names from your data source at runtime. When you generate a DataSet, it retrieves everything you specified in your Select statement in the Data Adapter Configuration dialog. You can limit your columns by changing the Select statement and creating a new DataSet.

## To use DataSets

---

1. Generate a DataSet.
2. Add multiple tables to a DataSet.
3. Define primary keys for DataTables in the DataSet.
4. Define column properties for your DataSet columns.
5. Define constraints for your columns.
6. Define relationships between tables in your DataSet.

## To generate a DataSet

---

1. From the *Data Explorer*, select a data source.
2. Drill down in the tree, then drag and drop the name of a table onto your Windows Form or Web Form.

This creates the BdpDataAdapter and BdpConnection for that data source and displays icons for those objects in the *Component Tray*.

**Note:** You can also drag a data source only onto the form, rather than a table, but in that case, Delphi 8 for .NET creates only a connection object for you. You must still create and configure the BdpDataAdapter object explicitly.

3. Click the BdpDataAdapter icon (named bdpDataAdapter1, by default) to select it.
4. Click the *Configure Data Adapter...* designer verb in the *Designer Verb area* at the bottom of the *Object Inspector*.

This displays the *Data Adapter Configuration* dialog.

5. If the SQL statement that is pre-filled on the dialog is acceptable, click the *DataSet* tab, otherwise, modify the SQL statement, then click the *DataSet* tab.
6. Select the *New DataSet* radio button.

**Tip:** You can accept the default name or change the name of the DataSet.

7. Click *OK* to generate the DataSet.

A DataSet icon appears in the *Component Tray* indicating that your DataSet has been created.

**Note:** By reviewing the code for the DataSet in the Code Editor, you can see that the columns are defined as generic dataColumns, whose columnName properties are assigned the value of the column name from the database table. This differs from how a typed DataSet is constructed, wherein the object name is constructed from the actual database column name, rather than assigned as a property value.

## To add multiple tables to one DataSet

---

1. From the *Data Explorer*, select a data source.
2. Drill down in the tree, then drag and drop the names of multiple tables, one at a time, onto your Windows Form or Web Form.

This creates the BdpDataAdapter for each table and one BdpConnection for that data source and displays icons for those objects in the *Component Tray*.

3. Click the BdpDataAdapter icon (named bdpDataAdapter1, by default) to select it.
4. Click the *Configure Data Adapter...* designer verb in the *Designer Verb area* at the bottom of the *Object Inspector*.

This displays the *Data Adapter Configuration* dialog.

5. If the SQL statement that is pre-filled on the dialog is acceptable, click the *DataSet* tab, otherwise, modify the SQL statement, then click the *DataSet* tab.
6. Select the *New DataSet* radio button.

**Tip:** You can accept the default name or change the name of the DataSet.

7. Click *OK* to generate the DataSet.

A DataSet icon appears in the *Component Tray* indicating that your DataSet has been created.



8. Repeat the Data Adapter configuration for each of the other data adapters, but select *Existing Data Set* on the *DataSet* tab when generating the DataSets for all data adapters except the first one you configure.

This generates a DataTable for each data adapter and stores them all in one DataSet.

**Note:** It is also possible to generate multiple DataSets, either one for each data adapter, or combinations of DataTables.

## To define primary keys for each DataTable in the DataSet

---

1. Select each data adapter in turn and set the Active property under *Live Data* in the *Object Inspector* to **True**.
2. Select the DataSet in the *Component Tray*.
3. In the *Object Inspector*, in the Tables property, click the ellipsis (...) button.

This displays the *Tables Collection Editor*. If you have set all of the data adapters' Active properties to **True**, the *Tables Collection Editor* will contain one member for each DataTable stored in the corresponding DataSet.

4. Select a table from the members list.
5. In the Primary Key field in the Table Properties, click on the DataColumn[] entry to display a pop-up list of column names.
6. Click the gray checkbox next to the column name of the column or columns that comprise the Primary Key.  
The number **1** appears in the gray checkbox when selected.
7. Define Column properties and Constraints for your Primary Key columns.

## To define column properties for your DataSet columns

---

1. In the Tables Collection Editor, click the (Collections) entry next to Columns in the Table Properties pane.  
This displays the Columns Collection Editor for the selected column.
2. Set the property values for the individual columns.

3. Repeat the process for each column.

## To define constraints for your columns

---

1. In the *Tables Collection Editor*, click the (Collections) entry next to Constraints in the *Table Properties* pane.

This displays the Constraints Collection Editor for the selected column.

2. Click *Add* to add either a Unique Constraint or a Primary Key Constraint.
3. If you selected Unique Constraint, the Unique Constraint dialog appears. Select one or more of the displayed column names. You can also select the Primary Key checkbox if you want to set the column as a primary key.

By setting the Unique Constraint on a column, you are enforcing the rule that all values in the column must be unique. This is useful for columns that contain identification numbers, such as employee numbers, social security numbers, part numbers, and so on.

**Note:** If you have already defined a primary-foreign key relationship between two tables, you may not be able to set a column as a primary key, based on the fact that it may already be set as the primary key, or based on a conflict with another relationship.

4. If you selected Foreign Key Constraint, the Foreign Key Constraint dialog appears. Select the tables you want to relate by choosing them from the Parent table and Child table drop down lists.
5. Click Key Columns to select the primary key column from the list.
6. Click Foreign Key Columns to select the foreign key column from the list.

**Warning:** The primary key and foreign key columns must have the same data type and must contain unique values. Columns that can contain duplicates are not good choices for primary or foreign keys. It is common to choose the same column name from each table for your primary-foreign key relationship.

## To define relationships between tables in the DataSet

---

1. Once you have defined primary keys for each DataTable, select the DataSet in the *Component Tray* if it is not already selected.
2. Click the ellipsis (...) button next to the Relations property in the *Object Inspector*.

This displays the blank *Relations Collection Editor* dialog.

3. Click *Add*.

This displays the *Relation editor* dialog

4. From the Parent table and Child table dropdown lists, choose the tables you want to relate.
5. Click the Key Columns field to choose a Primary Key column from the list of column names from the parent table.
6. Click the Foreign Key Columns field to choose a Foreign Key column from the list of column names from the child table.

**Note:** If you have already performed this procedure while setting constraints for your DataTables, you may find that all of the appropriate values are already established.

**Warning:** The primary key and foreign key columns must have the same data type and must contain unique values. Columns that can contain duplicates are not good choices for primary or foreign keys. It is common to choose the same column name from each table for your primary-foreign key relationship.

7. Click *OK*.
8. Repeat the process to define additional relations between the same DataTables.

# Using Typed DataSets

---

Typed DataSets provide certain advantages over standard DataSets. For one thing, they are derived from an XML hierarchy of the target database table. The XML file containing the DataSet description allows the system to provide extensive code-completion capabilities not available when using standard DataSets. Strong typing of DataSet methods, properties, and events allows compile-time type checking, and can provide a performance improvement in some applications.

## To create a strongly typed DataSet

---

1. From the Database Explorer, select the data source you want to use.
2. Drag and drop the name of the database table you want to use onto your form.  
This displays a BdpConnection icon and a BdpDataAdapter icon in the *Component Tray*.
3. Select the BdpDataAdapter.
4. Click the *Configure Data Adapter...* designer verb in the *Designer Verb* area beneath the *Object Inspector*.  
This displays the *Data Adapter Configuration* dialog.
5. Modify the pre-filled SQL statement if you like.
6. Click OK.  
**Note:** Do not create a DataSet by selecting the *DataSet tab* in the *Configure Data Adapter...* dialog. That tab applies only to standard DataSets.
7. Click the *Generate Typed Dataset...* designer verb in the *Designer Verb* area beneath the *Object Inspector*.  
This displays the *Generate Dataset* dialog.
8. Select the database table you want to use.
9. Click OK.  
This creates an instance of the typed DataSet and displays an icon *<DataSet Name>1* in the *Component Tray*. For example, if your DataSet is *DataSet1*, the new instance will be named *dataSet11*. You will also see that an XML .xsd file and a new program file appear in the *Project Manager* under your project.

## To modify how columns appear

---

1. After you have created a new typed DataSet, drop a *DataGrid* component onto your form.
2. Set the *DataSource* property to point to the typed DataSet and the *DataMember* property to point to the target table.
3. Click the *(Collection)* entry next to the *TableStyles* property.  
This displays the *DataGridTableStyle Collection Editor*.
4. Click *Add* to add a new member to the members list.
5. Click the drop down list next to the *MappingName* property.
6. Click the *(Collection)* entry next to the *GridColumnStyles* property.  
This displays the *DataGridColumnStyle Collection Editor*.
7. Click *Add* to add a new item to the members list.  
**Note:** By default the item is created as a Text Box Column. You can also expand the *Add* button and select the *BoolColumn* if you want a boolean.
8. Click the *MappingName* property, select the column you want to display in your grid, then change any additional properties you want, including the header name that will appear as the column header in the runtime grid.
9. Click *OK* twice.

**Note:** When you build and run the application, only the columns that you explicitly defined by following the steps in this procedure appear.

## To modify the structure of the dataset

---

1. In the *Project Manager*, double-click the *.xsd* file that contains the XML definition of your dataset.
2. Edit the XML file to reflect how you want the dataset to be structured.  
You can change data types, names, and anything else about the structure.
3. If you have the program code file (<dataset>.pas) open in the *Code Editor*, close it now.
4. Choose **Project** ► **Compile** to recompile the *.xsd* file.

If you re-open the program code file, you will see that the file contains the changes you made to the XML in the *.xsd* file.

## To set the Namespace property for a dataset

---

1. In the *Project Manager*, double-click the *.xsd* file that contains the XML definition of your dataset.
2. Find the `targetNamespace` property.
3. Change the text:

`http://www.changeme.now/DataSet1.xsd`

to a relevant namespace.

4. If you have the program code file (`<dataset>.pas`) open in the *Code Editor*, close it now.
5. Choose Project ► Compile to recompile the *.xsd* file.

If you re-open the program code file, you will see that the `InitClass()` class now contains the new namespace.

# Debugging Applications

---

# Adding a Watch

---

Add a watch to track the values of program variables or expressions as you step over or trace into code. Each time program execution pauses, the debugger evaluates all the items listed in the *Watch List* window and updates their displayed values.

You can organize watches into groups. When you add a watch group, a new tab is added to the *Watch List* window and all watches associated with that group are shown on that tab. When a group tab is displayed, only the watches in that group are evaluated during debugging. By grouping watches, you can also prevent out-of-scope expressions from slowing down stepping.

## To add a watch

---

1. Choose Run ► Add Watch to display the *Watch Properties* dialog box.
2. In the *Expression* field, enter the expression you want to watch.  
  
An expression consists of constants, variables, and values contained in data structures, combined with language operators. Almost anything you can use as the right side of an assignment operator can be used as a debugging expression, except for variables not accessible from the current execution point.
3. Optionally, enter a name in the *Group Name* field to create the watch in a new group, or select a group name from the list of previously defined groups.
4. Specify other options as needed (click *Help* on the *Watch Properties* dialog for a description of the options). For example, you can request the debugger to evaluate the watch, even if doing so causes function calls, by selecting the *Allow Function Calls* option.
5. Click *OK*.

The watch is added to the *Watch List* window.



# Attaching to a Running Process

---

You can attach to a process that is running on your computer or on a remote computer. This is useful for debugging a program that was not created with Delphi 8 for .NET.

## To attach to a running process

---

1. Choose Run ► Attach to Process to display the *Attach to Process* dialog box.
2. If the process is running on a remote computer, enter the name the computer in the *Remote Machine* field.


**Note:** The remote debug server must be running on the remote computer.

3. Select a process from the list of *Running Processes*.
4. Click *Attach*.

**Note:** If you want the process to pause after you have attached to it, check the *Pause After Attach* checkbox.

# Setting and Modifying Breakpoints

---

Breakpoints pause program execution at a certain location or when a particular condition occurs. You can set breakpoints in the *Code Editor* before and during a debugging session. During a debugging session, any line of code that is eligible for a breakpoint is marked with a blue dot  in the left gutter of the *Code Editor*.







## To set a breakpoint

---

1. Click the left gutter of the *Code Editor* next to the line of code where you want to pause execution.
2. To add a breakpoint and set options for the breakpoint, choose **Run ► Add Breakpoint ► Source Breakpoint** to display the *Add Source Breakpoint* dialog box.

**Tip:** To widen the *Code Editor* gutter, choose **Tools ► Options ► Editor Options ► Display** and increase the *Gutter width* option.

The following icons are used to represent breakpoints in the *Code Editor* gutter.

Icon	Description
	The breakpoint is valid and enabled. The debugger is inactive.
	The breakpoint is valid and enabled. The debugger is active.
	The breakpoint is invalid and enabled. The breakpoint is set at an invalid location, such as a comment, a blank line, or invalid declaration.
	The breakpoint is valid and disabled. The debugger is inactive.
	The breakpoint is valid and disabled. The debugger is active.
	The breakpoint is invalid and disabled. The breakpoint is set at an invalid location.

Breakpoints are displayed in the *Breakpoint List* window.

## To modify a breakpoint

---

1. Right-click the breakpoint icon and choose **Breakpoint Properties**.
2. Set the options in the *Source Breakpoint Properties* dialog box to modify the breakpoint.  
  
For example, you can set a condition, create a breakpoint group, or determine what action occurs when execution reaches the breakpoint.
3. Click *Help* for more information about the options on the dialog box.

4. Click *OK*.

## To create a breakpoint group

---

1. Right-click the breakpoint icon and choose *Breakpoint Properties*.
2. Enter a group name in the *Group* field, or select a name from the drop down list box to add the breakpoint to an existing group.
3. Click *OK*.

## To enable or disable a breakpoint or breakpoint group

---

1. Right-click the breakpoint icon in the *Code Editor* or in the *Breakpoint List* window and choose *Enabled* to toggle between enabled and disabled.
2. To enable or disable all breakpoints, right-click a blank area (not on a breakpoint) in the *Breakpoint List* window and choose *Enable All* or *Disable All*.
3. To enable or disable a breakpoint group, right-click a blank area (not on a breakpoint) in the *Breakpoint List* window and choose *Enable Group* or *Disable Group*.

Disabling a breakpoint or breakpoint group prevents it from pausing execution, but retains the breakpoint settings, so that you can enable it later.

## To create a conditional breakpoint

---

1. Choose *Run* ► *Add Breakpoint* ► *Source Breakpoint* to display the *Add Source Breakpoint* dialog box.
2. In the *Line number* field, enter the line in the *Code Editor* where you want set the breakpoint.  
  
To pre-fill the *Line number* field click a line in the *Code Editor* to pre-fill the *Line number* field prior to opening the *Add Source Breakpoint* dialog box.
3. In the *Condition* field, enter a conditional expression to be evaluated each time this breakpoint is encountered during program execution.

4. Click *OK*.

Conditional breakpoints are useful when you want to see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

If the conditional expression evaluates to true (or not zero), the debugger pauses the program at the breakpoint location. If the expression evaluates to false (or zero), the debugger does not stop at the breakpoint location.

## To associate actions with a breakpoint

---

1. Choose **Run ► Add Breakpoint ► Source Breakpoint** to display the *Add Source Breakpoint* dialog box.  
  
Alternatively, right-click the breakpoint icon and choose **Breakpoint Properties** to display the *Source Breakpoint Properties* dialog box.
2. Click *Advanced* to display additional options at the bottom the dialog box.
3. Check the actions that you want to occur when the breakpoint is encountered. For example, you can specify an expression to be evaluated and write the result of the evaluation to the *Event Log*.
4. Click *OK*.

## To change the color of the text at the execution point and breakpoints

---

1. Choose **Tools ► Options ► Editor Options ► Color**.
2. In the code sample window, select the appropriate language tab. For example, to change the breakpoint color for Delphi 8 for .NET code, select the **Delphi 8 for .NET** tab.
3. Scroll the code sample window to display the execution and breakpoint icons in the left gutter of the window.
4. Click anywhere on the execution point or breakpoint line that you want to change.
5. Use the *Foreground Color* and *Background Color* drop-down lists to change the colors associated with the selected execution point or breakpoint.

6. Click OK.

# Inspecting and Changing the Value of Data Elements

---

The *Debug Inspector* lets you inspect data elements by automatically formatting the type of data it is displaying. The *Debug Inspector* is especially useful for examining compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in the *Debug Inspector*, you can “walk” through compound data objects by opening a *Debug Inspector* on a component of the compound object.

**Note:** The *Debug Inspector* is only available when the process is stopped in the debugger.

## To inspect a data element directly from the Code Editor

---

1. In the *Code Editor*, place the insertion point on the data element that you want to inspect.
2. Right-click and choose **Debug ▸ Inspect** to display the *Debug Inspector*.

## To inspect a data element from the menu

---

1. Choose **Run ▸ Inspect** from the menu bar to display the *Inspect* dialog box.
2. In the *Inspect* dialog box, type the expression you want to inspect.
3. Click **OK**. The *Debug Inspector* is displayed.

Unlike watch expressions, the scope of a data element in the *Debug Inspector* is fixed at the time you evaluate it. If you use the *Inspect* command from the *Code Editor*, the debugger uses the location of the insertion point to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements that are not within the current scope of the execution point.

If you use **Run ▸ Inspect** from the menu, the data element is evaluated within the scope of the execution point. If the execution point is in the scope of the expression you are inspecting, the value appears in the *Debug Inspector*. If the execution point is outside the scope of the expression, the value is undefined and the *Debug Inspector* becomes blank.

## To change the value of a data element

---

1. In the *Debug Inspector*, select a data element that has an ellipsis (...) next to it. The ellipsis indicates that the data element can be modified.
2. Click the ellipsis (...), or right-click the element and choose **Change**.

3. Type a new value, then click *OK*.

## To inspect local variable values

---

1. While running in Debug mode, double-click any variable that appears in the *Local Variables* window.  
This displays the *Debug Inspector* for that local variable.
2. Click the *Data tab* to view strings, boolean values, and other values for such things as variable name, expression, and owner.  
**Tip:** If you want to drill down even deeper, to see the hexadecimal representation of a string, for instance, double-click the string value in the *Debug Inspector*.
3. Click the *Methods tab* to view all of the methods that have executed up to this point in the code.  
**Tip:** If you want to see the return type for any method, select the method and look at the status bar of the *Debug Inspector*, where the syntax line for the method, including the return type is displayed.
4. Click the *Properties tab* to view all of the properties for the active object, for instance, the form.
5. Click any property name to see its type displayed in the status bar of the *Debug Inspector*.
6. Click the question mark (?) icon to see the actual value for that property at this point of the execution of the application.

# Resolving internal errors

---

The error message, "Internal Error: X1234" indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process.

**Tip:** Internal error numbers indicate the file and line number in the compiler where the error occurred. This information may help Technical Support services track down the problem. Be sure to jot down this information and include it with your internal error description.

## To resolve an internal error

---

1. If the error occurs immediately after you have modified code in the editor, go back to the place where you made your changes and make a note of what was changed.
2. If you can undo or comment out the change and then recompile your application successfully, it is possible that the programming construct that you introduced exposed a problem with the compiler. If so, skip to "Review the code" below.

## If the problem still exists

---

1. Delete all of the .dcuil files associated with your project.
2. Close your project completely using FileClose All, then reopen your project, this will clear the unit cache maintained in the IDE. Alternatively, you can close the IDE and restart.
3. Another option is to try and recompile your application using the ProjectBuild option so that the compiler will regenerate all of your dcuils.
4. If the error is still present, exit the IDE and try to compile your application using the command line version of the compiler (dccil.exe) from a command prompt. This will remove the unit caching of the IDE from the picture and could help to resolve the problem.

## Review your code at the last modification point

---

1. If the problem still exists, go back to the place where you last made modifications to your file and review the code.

Typically, most internal errors can be reproduced with only a few lines of code and frequently the code involves syntax or constructs that are rather unusual or unexpected. If this is the case, try



modifying the code to do the same thing in a different way. For example, if you are typecasting a value, try declaring a variable of the cast type and do an assignment first.

### Examples

```
begin
    if Integer(b) = 100 then...
end;
var
    a: Integer;
begin
    a := b;
    if a = 100 then...
end;
```

Here is an example of unexpected code that you can correct to resolve the error:

```
var
    A : Integer;
begin
    { Below the second cast of A to Int64 is unnecessary; removing it
can avoid the Internal Error. }
    if Int64(Int64(A))=0 then
end;
```

2. In this case, the second cast of A to an Int64 is unnecessary and removing it corrects the error. If the problem seems to be a "while...do" loop, try using a "for...do" loop instead. Although this does not actually solve the problem, it may help you to continue work on your application.

If this resolves the problem, it does not mean that either "while" loops or "for" loops are broken but more likely it means that the manner in which you wrote your code was unexpected.

3. Once you have identified the problem, we ask that you create the smallest possible test case that still reproduces the error and submit it to Borland.

## Other techniques for resolving internal errors

---

1. If error seems to be on code contained within a while...do loop try using a for...do loop instead or vice versa.

2. If it uses a nested function or procedure (a procedure/function contained within a procedure/function) try "unnesting" them.
3. If it occurs on a typecast look for alternatives to typecasting like using a local variable of the type you need.
4. If the problem occurs within a with statement try removing the with statement altogether.
5. Try turning off compiler optimizations under ProjectOptions ► Compiler.

## When all else fails

---

1. Typically, there are many different ways to write any single piece of code. You can try and resolve an internal error by changing the code. While this may not be the best solution, it may help you to continue to work on your application. If this resolves the problem, it does not mean that either "while" loops or "for" loops are broken but perhaps that the manner in which you've written your code was unexpected and therefore resulted in an error.
2. If you've tried your code on the latest release of the compiler and it is still reproducible, create the smallest possible test case that will still reproduce the error and submit it to Borland. If it is not reproducible on the latest version, it is likely that the problem has already been fixed.

## Configuring the IDE to avoid internal errors

---

1. Create a single directory where all of your .dcpil files (precompiled package files) are placed. For example, create a directory called C:\DCPIL and under ToolsEnvironment Options select the Library tab and set the DCPIL output directory to C:\DCPIL. This setting will help ensure that the .dcpil files the compiler generates are always up-to-date which is particularly useful when you move a package from one directory to another. You can create a .dcuil directory on a per-project basis using ProjectOptions ► Directories/Conditionals ► Unit output directory.
2. The key here is that you always want to be using the most up-to-date versions of your .dcuil and .dcpil files. Otherwise, you may encounter internal errors that are easily avoidable.

# Modifying Variable Expressions

---

After you have evaluated a variable or data structure item, you can modify its value. When you modify a value through the debugger, the modification is effective for the program run only. Changes you make through the *Evaluate/Modify* dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the *Code Editor*, then recompile your program.

## To change the value of an expression

---

1. Choose Run ► Evaluate/Modify.
2. Specify the expression in the *Expression* edit box.

To modify a component property, specify the property name, for example, `this.button1.Height` `Self.button1.Height`.

3. Enter a value in the *New Value* edit box.

The expression must evaluate to a result that is assignment-compatible with the variable you want to assign it to. Typically, if the assignment would cause a compile or runtime error, it is not a legal modification value.

4. Choose *Modify*. The new value is displayed in the *Result* box.

You cannot undo a change to a variable after you choose *Modify*. To restore a value, however, you can enter the previous value in the *Expression* box and modify the expression again.

**Note:** You can change individual variables or elements of arrays and data structures, but you cannot change the contents of an entire array or data structure with a single expression.

**Warning:** Modifying values (especially pointer values and array indexes), can have undesirable effects because you can overwrite other variables and data structures. Use caution whenever you modify program values from the debugger.

# Preparing a Project for Debugging

---

## To activate the integrated debugger

---

1. Choose Tools ► Options ► Debugger Options.
2. Select the *Integrated Debugging* option.
3. Click *OK*.
4. Optionally review the settings on the other debugging pages.

## To generate debug information for a project

---

1. Choose Project ► Options ► Compiler.
2. Select Directories/Conditionals.
3. Enter the path for the *Debug Source Path*.
4. Click *OK*.

The program database file (.pdb), which contains the debugging information for the application, will be created in the source directory the next time you compile the project.

# Debugging Remote Applications

---

Remote debugging lets you debug a Delphi 8 for .NET application running on a remote computer. Once the remote debug server is running on the remote computer, you can use Delphi 8 for .NET to connect to that computer and begin debugging.

## Prerequisites and security considerations for remote debugging

- The local and remote computers must be connected through TCP/IP.
- All of the files required for debugging the application must be available on the remote computer before you begin debugging. This includes executables, dlls, assemblies, data files, and PDB (debug) files.
- In addition to the port that the remote debug server listens on, a connection is opened for each application that is being debugged. Additional port numbers are chosen dynamically by Windows; a firewall that only allows connections to the listening port will prevent the remote debugger from working.

**Warning:** The connection between Delphi 8 for .NET and the remote debug server is a simple TCP/IP socket, with neither encryption nor authentication support. Therefore, the remote debug server should not be run on a computer that can be accessed over the network by untrusted clients.

## To install and start the remote debug server

---

1. Skip to step 4 if Delphi 8 for .NET is installed on the remote computer, since the remote debug server (dbkwmc71.exe) is already available, by default, at C:\Program Files\Borland\BDS\2.0\Bin.
2. Copy dbkwmc71.exe and dbkpro71.dll from the Delphi 8 for .NET \bin directory on your local computer to the directory of your choice on the remote computer.
3. On the remote computer, register dbkpro71.dll by running the regsvr32.exe registration utility. For example, on Windows XP, enter C:\Windows\System32\regsvr32.exe dbkpro71.dll at the command prompt.
4. On the remote computer, run dbkwmc71.exe using the following syntax:

```
dbkwmc71.exe [-listen [hostname:]port]
```


where:

hostname is an optional host name or TCP/IP address for binding to a particular host, for example, somehost or 127.0.0.1. If you specify hostname, you must also specify :port.

port is an optional (required if hostname is specified) port number or standard protocol name, for example, 8000 or ftp. If omitted, 64447 is used as the port number.

Examples:

```
dbkwmc71.exe
dbkwmc71.exe -listen 8000
dbkwmc71.exe -listen somehost:8000
dbkwmc71.exe -listen 127.0.0.1:8000
```

After the remote debug server is started, its icon  appears in the Windows taskbar.


## To connect to the remote computer and start debugging

---

1. On your local computer, open the Delphi 8 for .NET project that corresponds to the application to be debugged.
2. Choose Project ► Options ► Debugger ► Remote.  
Alternatively, choose Run ► Parameters ► Debugger ► Remote.
3. In the *Remote Path* field, enter the path for the application's executable file on the remote computer. The remote debug server will use this path to find the executable, so specify a path relative to the directory that contains dbkwmc71.exe.
4. In the *Remote Host* field, enter the host name or TCP/IP address of the remote computer.  
If a port was specified when starting dbkwmc71.exe, enter a colon after the host name, followed by the port, for example, somehost:8000 or 127.0.0.1:8000. Otherwise, the default port 64447 will be used.
5. Check the *Debug project on remote computer* checkbox and click *OK*.
6. Choose Run ► Run to begin the remote debugging session. This will require more time than starting a local debugging session.
7. Debug the application as you would normally.
8. Optionally, if you want to prevent other computers from connecting to the remote computer while you are debugging, the remote debug server can be shut-down without affecting your connection. See the following section.
9. When you are done debugging, choose Run ► Program Reset to end the debugging session and terminate your connection to the remote computer.

## To shut down the remote debug server

---

1. On the remote computer, in the Windows taskbar, right-click the  *Borland Remote Debugger Listener* icon.
2. In the pop-up menu, choose Exit.

Shutting down the remote debug server does not affect active debugging sessions.

## Editing Code

---



# Using Code Folding

---

Code folding lets you collapse (hide) and expand (show) your code to make it easier to navigate and read.

## To collapse and expand code

---

1. In the *Code Editor*, click the plus(+) sign to the left of a code block to collapse the code.
2. Click the minus (-) sign to expand the code block.

**Tip:** To turn off code folding for the current edit session, press and hold Ctrl+Shift and then K and then O. To collapse the nearest code block, press and hold Ctrl+Shift and then K and E. To expand the nearest code block, press and hold Ctrl+Shift and then K and U. To expand all code, press and hold Ctrl+Shift and then press K and A.

# Customizing Code Editor

---

Borland Delphi 8 for .NET lets you customize your *Code Editor* by using the available settings to modify keystroke mappings, fonts, margin widths, colors, syntax highlighting, and indentation styles.

## To customize general Code Editor options

---

1. Choose Tools ► Options.
2. Scroll down to *Editor Options*.
3. Select any of the customization options to make modifications to the *Code Editor*.



# Recording a Keystroke Macro

---

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

## To record a macro

---


1. In the *Code Editor*, click the record macro button  at the bottom of the code window to begin recording.
2. Type the keystrokes that you want to record.
3. When you have finished typing the keystroke sequence, click the stop recording button .
4. To record another macro, repeat the previous steps.

**Note:** Recording a macro replaces the previously recorded macro.

The macro is now available to use during the current IDE session.

## To run a macro

---

1. In the *Code Editor*, position the cursor in the code where you want to run the macro.
2. Click the macro playback button  to run the macro.

If the button is dimmed, no macro is available.

# Using Code Insight

---

*Code Insight* is a set of features in the *Code Editor* that provide code completion, display code parameter lists, and tool tips for expressions and symbols.

## To enable Code Insight

---

1. Choose Tools ► Options.  
The *Options* dialog box appears.
2. Under *Editor Options*, select *Code Insight*.
3. Review and set the options and color preferences as needed.
4. Click OK.

## To use Code completion

---

1. Choose Tools ► Options.  
The *Options* dialog box appears.
2. Select *Code Insight* and enable the *Code Completion*.
3. On the *Code Editor*, type an object or class name followed by a dot (.) to display a list of types, properties, methods, and events.
4. Select the one appropriate for the class and press ENTER.

## To use Code parameters

---

1. Choose Tools ► Options.  
The *Options* dialog box appears.
2. Select *Code Insight* and enable the *Code parameters* checkbox.

3. In the *Code Editor*, type a method name and an open parenthesis to display the syntax for the method's arguments.

## To use ToolTip expression evaluation

---

1. Choose Tools ► Options.  
The *Options* dialog box appears.
2. Select *Code Insight* and enable the *ToolTip expression evaluation* checkbox.
3. On the *Code Editor*, point to any variable to display its current value while your program has paused during debugging.

## To use ToolTip symbol insight

---

1. Choose Tools ► Options.  
The *Options* dialog box appears.
2. Select *Code Insight* and enable the ToolTip symbol insight.
3. On the *Code Editor*, point to any identifier to display its declaration while editing your code.

# Using Code Snippets

---

Code snippets are reusable code statements that are accessible from the *Tool Palette*. While using the *Code Editor*, you can insert predefined code snippets into your code or add your own code snippets to the *Tool Palette*.

## To add a code snippet to your code

---

1. In the *Code Editor*, select a code snippet from the *Tool Palette*.
2. Double-click the selected code snippet or drag and drop the code snippet onto the *Code Editor* to include as part of your code.

## To add a code snippet to the Tool Palette

---

1. In the *Code Editor*, type the code that you want to save as a code snippet.
2. Select the code that you just typed.
3. Press and hold the ALT key and drag and drop the code onto the *Tool Palette*.



# Adding Columns to a Component

---

There may be situations in which you want to add columns to components at runtime, for instance, to accommodate computed or derived values that have no corresponding design-time field in a class or in an underlying data source. The ECO framework provides the means to add columns at design-time, which will then appear in your component, such as a datagrid at runtime. This procedure assumes you have read the Modeling Tools Overview.

## To add a column to an expression handle

---

1. Assuming you have a form application, with a datagrid and an expression handle already defined, select the expression handle in the Component Tray.
2. Click the ellipsis (...) button on the Column field in the Object Inspector.  
This displays the Column Collection Editor.
3. Click *Add* to add a new column.
4. Click the ellipsis (...) button on the Expression field in the properties pane of the dialog.
5. Create your OCL expression by double-clicking objects in the right-hand textbox of the OCL Expression Editor and adding elements.  
For instance, double-click a class name, then double-click the `allInstances` expression to add it.
6. Click OK.
7. Click OK.
8. Compile the application.

If you are using a datagrid that references the expression handle for which you created a new column, the new column should appear in the design-time datagrid.



# Building an ECO-Enabled User Interface

---

It is assumed that you have a model with two classes, with a master/detail relationship between them. A common example of this relationship includes Customer and Order classes, where one Customer can have zero or many Orders. The procedure demonstrates the creation of two grids: one that displays records on the master side of the association, and a second grid that displays the associated detail records. The grids will be linked so that the detail grid always shows the objects associated with the current selection in the master grid. It is also assumed that you have configured your application's ECO Space with either an RDBMS persistence component, or an XML file persistence component. For more information on using the integrated class diagramming tools, and on configuring the application's ECO Space, please refer to the links at the end of this document.

In the ECO framework, the term "handle" means a value. A handle could represent a single value, such as the value of one attribute of an object. Or, it could represent a collection of values, such as a list of objects returned by an OCL expression. Handles implement the `System.ComponentModel.IListSource` interface, making them suitable for use as data sources in user interface components.

The *ECO Enabled Windows Form wizard* creates a basic Windows Form with two additional properties: A RootHandle (default name `rhRoot`), and an ExpressionHandle (default name `ehRoot`). The form's constructor is written to take a reference to your application's ECO Space as a parameter, and it includes code to set the ECO Space property of the RootHandle.

## To add a DataGrid for the master side of the relationship

---

1. Open the *WinForm designer* for the ECO-enabled Windows Form.
2. Select the ExpressionHandle (`ehRoot`), and set its RootHandle property to `rhRoot` using the *Object Inspector*.
3. While the ExpressionHandle is still selected, set its Expression property using the OCL expression editor. Using the example above, you might enter an OCL expression that results in a collection of `Customer` objects, such as `Customer.allInstances`.
4. Drag a DataGrid component from the *Tool Palette* onto the form.
5. Set the grid's DataSource property to the ExpressionHandle, `ehRoot`.

If you are using an RDBMS as the persistence method, the DataGrid will display column headings that correspond to the attributes you have defined on the Customer class. The column headings will not display at designtime when persisting objects to an XML file, but they will appear at runtime.

Continuing with the `Customer` example, to populate the grid with a new instance of the `Customer` class, execute code such as the following: `Customer newCustomer = new Customer (EcoSpace); Customer := Customer.Create(EcoSpace);`

Where the `EcoSpace` parameter is the instance of your application's ECO Space that is allocated in your application's main WinForm. The grid is automatically synchronized with the ECO Space; when you create a new instance within the ECO Space, a new row is added to the grid.

Using the ECO framework it is quite simple to construct two grids that are linked together in a master/detail relationship as described above. Linking grids together requires the use of a `CurrencyManagerHandle` object, and a second `ExpressionHandle` object to retrieve the objects on the detail side of the relationship.

The `CurrencyManagerHandle` manages the synchronization between the `ExpressionHandle` for the master objects, and the `ExpressionHandle` for the detail objects. It is helpful to think of the `CurrencyManagerHandle` object as sitting between the two `ExpressionHandles`:

`EcoSpace <-- RootHandle <-- Master ExpressionHandle (master) <-- CurrencyManagerHandle <-- Detail ExpressionHandle`

## To configure a `CurrencyManagerHandle` in a master/detail relationship

---

1. Scroll the *Tool Palette* to the Enterprise Core Objects category.
2. Select the `CurrencyManagerHandle` component, and drag it to the ECO-enabled Windows Form.
3. Select the `ExpressionHandle` component, and drag it to the form. This `ExpressionHandle` will manage the detail side of the relationship.
4. Select the `CurrencyManagerHandle` component on the designer, and set its `RootHandle` property to the `ExpressionHandle` on the master side of the relationship.
5. Set the `CurrencyManagerHandle` component's `BindingContext` property to the `DataGrid` that displays the objects on the master side of the relationship.
6. Select the detail `ExpressionHandle` and set its `RootHandle` property to the `CurrencyManagerHandle` object.
7. Set the `Expression` property of the detail `ExpressionHandle` to an OCL expression that completes the expression on the master side of the relationship.

For example, if the `Customer/Order` relationship is expressed in an association called `CustomerOrders`, and `orders` is the name of the association end (or role), then the OCL

expression for the ExpressionHandle is simply `orders`. The complete OCL expression can be thought of as `Customer.allInstances.orders`.

Finally, you can complete the user interface by adding a second DataGrid to display the detail objects.

## **To add a DataGrid for the detail side of the relationship**

---

1. Drag a DataGrid from the *Tool Palette* to the form.
2. Set the DataGrid's DataSource property to the detail ExpressionHandle.

# Building Applications with the ECO Framework

---

Building an ECO-enabled application consists of a number of steps, each with its own set of procedures. This topic presents a grand overview of the entire process from start to finish.

## To Create an ECO Enabled Application

---

1. Create an ECO application using the *ECO Application Wizard* in the *New Items* dialog box. Alternatively, you can import a model in XMI format.
2. Create or edit your model using the integrated UML diagramming tools: The *Model View Window*, integrated UML class diagram, *Object Inspector*, and *Tool Palette*.
3. Configure your application's ECO Space. The ECO Space will contain instances of the classes in your model; it is a middle layer between your application's front-end, and the persistence layer.
4. Build a user interface for your application. You can connect data-aware .NET components to the objects in your ECO Space through element handles such as *ExpressionHandles*. The element handles implement the interfaces required in order to render their values in a data-aware component. You can use the *OCL Expression Editor* to enrich the specification of your model by adding invariant constraints and derived attributes.
5. Deploy your ECO enabled application.

# Deploying an ECO-Enabled Application

---

## To deploy an ECO application

---

1. Open your project and select View ► Project Manager to display the *Project Manager* window if it is not already open.
2. Select the appropriate compiler settings in the *Project Options* dialog box.

**Note:** You must set the appropriate build settings on each project in your application's project group.

3. Select Project ► Build <Project Name> where <Project Name> is the actual name of your project to build your application.

The build targets for each project in your application's project group will be generated per their own respective project settings. Referenced assemblies that have their Copy Local setting checked will be copied to the output directory of the project that references them.

In addition to the other assemblies your project references, there are five ECO-specific assemblies that must be deployed with all ECO applications (these assemblies are displayed under your project's *References* node in the *Project Manager* window):

---

Borland.Eco.Core.dll

---

Borland.Eco.Handles.dll

---

Borland.Eco.Interfaces.dll

---

Borland.Eco.Ocl.ParserCore.dll

---

Borland.Eco.Persistence.dll

---

The Delphi 8 for .NET installer deploys these five assemblies into the .NET Global Assembly Cache (GAC). The GAC cannot be viewed or manipulated directly however, and copies of these files are kept with other shared assemblies in Delphi 8 for .NET's Common Files folder. The default path to this location is \Program Files\Common Files\Borland Shared\BDS\Shared Assemblies\<version>, where <version> is the version number of Delphi 8 for .NET that is installed on the development machine.

On the end-user's machine, you can deploy the ECO assemblies into the GAC, or you can choose to deploy them into the application's installation directory. If you will be deploying multiple ECO applications however, it's best to deploy them as shared assemblies.

# Exporting a Code Visualization Diagram to an Image

---

You can export a code visualization diagram to an image and then open the image in any graphic viewer that supports the Windows bitmap (.bmp) file format.

## To export a diagram to an image

---

1. Open a project.
2. Click on the *Model View* tab.
3. Right-click on the diagram node name, which is *default*, by default.
4. Choose *Open Diagram*.
5. Right-click on the diagram surface.
6. Choose *Export to Image...* to display the *Export diagram to Image* dialog.
7. Adjust any zoom settings, if necessary.
8. Click *Save*.
9. Name the image and click *Save*.

# Importing and Exporting a Model Using XMI

---

Delphi 8 for .NET supports the XML Metadata Interchange (XMI), version 1.1. Please see the link to the OMG website at the end of this document for more information on XMI or to download the complete specification.

## To import a model in XMI format

---

1. Export the model from the modeling tool, using the XMI format. When exporting from Rational Rose, choose XML version 1.1, using the Unisys extension.
2. In Delphi 8 for .NET, start a new ECO application by selecting File ► New ► Other, and choosing *ECO Application* from the *New Items* dialog box.
3. Open the *Model View Window*, and right-click on the top-level project node in the tree. Choose Import Project from XMI.
4. In the *XMI Import* dialog box, click the *Browse...* button to navigate to the XMI file you exported in step one.
5. Click the *Import* button in the *XMI Import* dialog box.

Delphi 8 for .NET will generate ECO-enabled, Delphi source code for the model elements on the class diagrams in the XMI file.

## To export a model in XMI format

---

1. Open the *Model View Window* and right-click the top-level project node in the tree. Select Export Project to XMI.
2. In the *XMI Export* dialog box, select the XMI version, and XMI encoding appropriate for the tool you will ultimately use to open the model file.
3. Click the *Browse...* button to navigate to the destination folder and enter a target file name for the exported file.
4. Click the *Export* button in the *XMI Export* dialog box.

# Using the Model View Window and Code Visualization Diagram




Code Visualization allows you view and navigate the logical structure of your application, as opposed to the file-centric view of the *Project Manager*.

## To Display the Model View Window

1. Start a new project, or load an existing one.
2. Select View ► Model View.

The *Model View window* will open, showing the elements of your project in a tree view.

### Using the Model View window with Code Visualization diagrams

To...	...do this
View or hide nested elements within a UML package, class, or interface...	Click the plus sign (+) next to the element's icon to view nested items, or click the minus sign (-) to hide nested items.
View a <i>Code Visualization diagram</i> for a .NET namespace...	Expand the namespace icon (  ) and double-click the diagram icon (  ) in the <i>Model View tree</i> .
View a <i>Code Visualization diagram</i> for the entire project...	Expand the project icon (  ) in the <i>Model View tree</i> , and double-click the diagram icon.
Open the <i>source code editor</i> on a specific item in the <i>Model View tree</i> ...	Right-click on the item, and select Open Source from the menu. Note that a .NET namespace can span multiple source files. You can't open a source file for a namespace directly from the <i>Model View tree</i> .
Open the <i>Code Visualization diagram</i> on a specific item in the <i>Model View tree</i> ...	Right-click on the item, and select Show Element on Diagram from the menu.

The Code Visualization diagram has a set of functions that can help you view large models, show or hide attributes, properties, etc., and to move from the graphical depiction on the diagram, directly to the source code for that item.

### Using the Code Visualization diagram

To...	...do this
Rearrange items on the diagram...	Click the item and drag it to a new location.



View or hide the attributes, operations, properties, and nested types for an item on the diagram...	Click the plus sign (+) next to the category (attributes, operations, etc.) you want to view. Click the minus sign (-) to hide items of a particular category.
Perform an automatic layout of the items on the diagram...	Right-click anywhere in the <i>Code Visualization diagram</i> window and select Do Full Layout.
Print the diagram...	Right-click anywhere in the <i>Code Visualization diagram</i> window and select Print.
Open the <i>source code editor</i> on a specific item on the <i>Code Visualization diagram</i> ...	Right-click on the item, and select Open Source from the menu.
Save the diagram as an image...	Right-click anywhere in the <i>Code Visualization diagram</i> and select Save Image from the menu.

# Using the Model View Window and ECO Class Diagram

---

## To Display the Model View Window



---

1. Start a new ECO project using the *ECO Application wizard*, or load an existing one.

**Note:** The class diagramming and source code generation tools are only available in ECO applications. In other types of projects, you can use code visualization to generate a static structure diagram from Delphi source code, but you cannot modify the source code through the diagram.

2. Select View  Model View.

The *Model View window* will open, showing the elements of your project in a tree view.

The *Model View window* allows you to view both code visualization and ECO-enabled diagrams. All diagrams in the *Model View tree* are located within a container. Code visualization diagrams are contained within .NET namespace nodes (denoted by the  icon). ECO-enabled class diagrams are contained within UML package nodes (denoted by the  icon). To open any diagram, double-click its node in the *Model View tree*.

There are two ways to add an element to an ECO class diagram: The *Tool Palette*, or the right-click context menu. Context menus are available directly on the diagram surface, on elements already on the diagram, and on nodes in the *Model View tree*.

## To add an element from the Tool Palette

---

1. Select the type of element you wish to add by clicking once with the left mouse button.
2. Click on the diagram surface and drag a rectangle. The graphic representing the new element will be drawn to the size of the rectangle when you release the mouse button.

**Note:** To nest a UML package within an existing package, click the mouse once within the containing package symbol on the diagram (i.e. do not drag a rectangle). Similarly, to create a class within an existing package, click the mouse once within the UML package symbol.

## To add an element using the context menu

---

1. Right-click the mouse on the element (class diagram surface, UML package, or class) that will contain the new item.
2. Select either the Add New Item menuitem, or use the Add submenus to add the new element. The Add New Item selection will open a tabbed dialog in which you can select the kind of element you wish to add, and set the its name. Using the Add submenus will immediately add the new item to its container, with a default name that you can change later.

The kinds of elements available to add depends completely on the kind of container you have selected. For example, you can only add attributes and operations to classes. Similarly, you can only add classes, packages, and notes to an existing UML package.

**Note:** When you add an element directly to the diagram surface (as opposed to an existing package or class on the diagram), it will be contained within the UML package that owns the diagram.

## To create an association between two classes

---

1. Select Association as the type of element in the *Tool Palette*.
2. Move the mouse cursor over one of the classes to participate in the association. The class will be highlighted with a rectangle.
3. Click the mouse within the highlighted class and then move the mouse to the class at the other end of the association.
4. Again, the class will be highlighted with a rectangle. Click the mouse over the highlighted class.

An association will be drawn between the two classes. Select the association, and use the *Object Inspector* to set its properties. Association end properties such as the name and multiplicity are also configured using the *Object Inspector*.

## To add attributes and operations to classes

---

1. Select the class, either on the class diagram, or in the *Model View tree*.
2. Right-click the mouse and choose either the Add New Item menuitem, or use the Add submenus as described above.

3. Type the name of the attribute or operation. Use standard UML syntax to set the visibility, name and type (for attributes), or parameters and return type (for operations). The UML syntax for declaring attributes and operations is:

- Attributes: `{visibility}attrName:type[=expression]`
- Operations: `{visibility}opName(param1:type, param2:type, ...):returnType`

Where `{visibility}` is one of:

- `+` : Public visibility
- `-` : Private visibility
- `#` : Protected visibility

When declaring attributes you can include an optional expression, which will become the initial value of the attribute. For example, the declaration: `+myAttribute:int=17` declares a public attribute named `myAttribute`, of type `int`, which will have an initial value of 17.

An attribute can be made to be "derived" by prefixing its declaration string with a slash "/", or by selecting it and changing the `Derived` property to `true` in the *Object Inspector*. When you create a derived attribute, you will also need to open the source code for that attribute (right click the attribute and select the Open Source menuitem), and remove the `[EcoAutoMaintained]` .NET attribute. Otherwise, any code you write in the attribute's `get/set` methods will be overwritten when the IDE regenerates source code.

**Note:** Like all elements on the class diagram, attributes and operations can be individually selected and then configured using the *Object Inspector*.

# Using the ECO Space Designer

---

An ECO Space is a container for the runtime instances of the classes in your model. The ECO Space designer lets you select UML packages from your model, choose the persistence mechanism for objects, create or evolve the database schema, and perform designtime validation of the model.

You cannot work directly with the class `Borland.Eco.Handles.EcoSpace`. Instead, the IDE automatically creates a subclass of the `EcoSpace` class for you, when you create a new ECO application. If you have imported a model from another tool, such as Bold for Delphi or Together Control Center, you can add an ECO Space to your project using the *ECO Space Wizard* in the *New Items* dialog box.

Your application's ECO Space is implemented in one source file. The default source file name is `EcoSpace.pas`. To open the *ECO Space designer* and begin work, double-click the source code file in the *Project Manager* window, and then click the *Design* tab. This document first describes the basic procedure for configuring an ECO Space. Each step is then explained in more detail in the following sections.

**Warning:** You must rebuild your application prior to using the *ECO Space designer*. The ECO framework makes extensive use of .NET custom attributes, and building your application ensures that the designer is working with the correct assembly metadata.

## To configure an ECO Space using the designer


---

1. Select the UML packages containing the classes that you want to be persistent.
2. Choose a persistence method, either an RDBMS, or XML file. Here you will add the required persistence components to the ECO Space, and then configure the ECO Space to use the chosen persistence method by setting properties in the *Object Inspector*.
3. Validate the Model. This will cause the IDE to perform a number of checks to make sure the model is well-formed. For example, OCL expressions are checked to make sure they are valid.
4. If you are using an RDBMS, create an empty database. The exact procedure will vary depending on your database vendor. This procedure will use an Interbase database as an example.
5. Add a connection handle component to the ECO Space and configure its connection string.
6. If you are using an RDBMS and you are starting from scratch, create the initial database schema by clicking on the *Create Database Schema* button. Otherwise, if you have made changes to the model, or you want to add or remove a UML package, you can use the *Evolve Db* button to update the existing database schema.

**Note:** When a class or attribute is deleted and the database schema evolved, the corresponding columns are removed from the database and the data is lost. The IDE will warn you if this is the case, giving you a chance to cancel the operation.

## To select UML packages

---

1. Click the *Select Packages* button (  ) on the designer. The *Select packages* dialog box will open.
2. A full list of available UML packages is shown in the *Available Packages* list box. UML packages that are already managed in the ECO Space are shown in the *Selected Packages* list box.
3. To add a single UML package, select it in the list, and click the '<' (left arrow) button. To add all available packages, click the '<<' (double left arrow) button.
4. To remove a single package from the ECO Space, select the package in the *Selected Packages* list, and click the '>' (right arrow) button. To remove all selected packages from the ECO space, click the '>>' button.

## To configure the ECO Space for the chosen persistence method

---

1. Scroll the *Tool Palette* so that the Enterprise Core Objects category is in view. There are three persistence methods to choose from:
  - PersistenceMapperBdp. Uses the Borland Data Provider classes for database connectivity.
  - PersistenceMapperSqlServer. Uses the native .NET database connectivity classes, which are optimized for use with Microsoft SQL Server.
  - PersistenceMapperXML. Objects will be stored into an XML file instead of a relational database.

**Tip:** It is often useful to store your objects in an XML file during initial development and prototyping, and then switch to a relational database as your model becomes more stable.

2. Drag the appropriate persistence mapper from the *Tool Palette* to the *ECO Space designer* surface.
3. Click the mouse on an empty portion of the *ECO Space designer* surface, so that the *Object Inspector* is showing the properties of the ECO Space.
4. Set the PersistenceMapper property to the persistence mapper you created in step 2.

**Note:** When using the PersistenceMapperXML component, it is not necessary to create or evolve a database schema as described below. When persisting to an XML file, all

that is required is to select the component on the ECO Space designer, and set its FileName property.

## To create an empty Interbase database using the IBConsole program

---

1. Start the Interbase Console program, IBConsole.
2. Login to the server where you want the new database to be created. Under the Databases node you will see a list of databases that reside on that server.
3. Right-click on the Databases node, and select Create Database.
4. Type the path and file name of the database. Customize any database parameters you wish to change, and click OK.

The new database will be displayed under the Databases node in the IBConsole window.

Next, you must go back to Delphi 8 for .NET and add a connection handle, and then connect it to the persistence mapper component.

## To add and configure a connection handle



---

1. Scroll the *Tool Palette* so that the Borland Data Provider category is in view.  
**Note:** If you are using a SQL Server persistence handle, scroll to the Data components category.
2. Drag a BdpConnection (or SqlConnection) component onto the ECO Space designer surface.
3. Select the persistence mapper component on the designer surface.
4. Set the Connection property of the persistence mapper component to the connection handle you created in step 2.
5. Set the default vendor-specific configuration settings of the persistence mapper. Do this by right-clicking on the persistence mapper component, and selecting the appropriate item from the context menu. For example, to set the default settings for an Interbase database, select Interbase|dialect 3 setup from the context menu.
6. Right-click the connection handle component, and select Connection Editor from the context menu. The ConnectionString will vary depending on the database vendor. For an Interbase database, you will need to edit the connection string to reflect the correct path to the database file. Default, vendor-

specific connection strings are available both from the *Connections Editor* dialog, and from the *ConnectionString* property's drop-down list in the *Object Inspector*.

## To create or evolve the database schema

---

1. If you are creating an ECO application from scratch, click the Create Database Schema button (  ) on the designer. This will create the tables necessary to support the classes and relationships you have created in the model.  
  
- or -
2. If you are working with an existing ECO application and you have made changes to the model, click the Evolve Database button (  ) on the designer.

**Note:** During creation or evolution of the database schema, the ECO tab in the *message pane* will display status messages and results of the operation.



# Using the ECO Wizards

---

This topic describes each ECO Wizard available in the *New Items* dialog box, and how each wizard is used during the creation and maintenance of an ECO-enabled application.

## ECO Wizards

---

1. Create a new ECO application from scratch.
2. Add an ECO-enabled Windows Form to your project.
3. Add a new UML package to your project.
4. Add a new subclass of the `Borland.Eco.Handles.EcoSpace` class.

## Creating a new ECO application

---

1. Click on File ► New ► Other to open the *New Items* dialog box.
2. Select the *Delphi for .NET Projects* project category.
3. Double-click the *ECO Application* icon. The *New Application* dialog box will appear.
4. Type the name of your project, and use the *Browse* ('...') button to navigate to the folder where you want the project files to reside.
5. Use the subsequent Save As dialogs to name your project files and save initial copies in the project folder.

The *ECO Application Wizard* generates a new project containing the following files:

CoreClasses.pas	Contains the source code for the UML packages, interfaces, classes and their associations, and all other types in your model.
EcoSpace.pas	Contains source code for the subclass of <code>Borland.Eco.Handles.EcoSpace</code> .
EcoWinForm.pas	Contains an ECO enabled WinForm. An ECO enabled form is a subclass of <code>System.Windows.Forms.Form</code> that provides a property for setting and retrieving the ECO Space. An instance of the application's ECO Space is passed to the form's constructor. A default root handle and expression handle are also provided, as is source code to connect the root handle to the ECO Space.

WinForm.pas	Contains source code for the application's main form. The main form for an ECO application provides a property that holds an instance of the application's ECO Space. The form also contains code to automatically allocate the ECO Space.
Borland.Eco.Core.dll	These are the .NET assemblies required by all ECO applications. The <i>ECO Application Wizard</i> automatically adds references to these assemblies, and they must be distributed with your application along with all other referenced assemblies.
Borland.Eco.Handles.dll	
Borland.Eco.Interfaces.dll	
Borland.Eco.Ocl.ParserCore.dll	
Borland.Eco.Persistence.dll	

## To Add an ECO Enabled Windows Form to Your Project

1. Click on File ► New ► Other to open the *New Items* dialog box.
2. Select the *ECO/Delphi for .NET Files* category.
3. Double-click the *ECO Enabled Windows Form* icon.

The *ECO Enabled Windows Form Wizard* generates a new subclass of `System.Windows.Forms.Form`. The new class provides a constructor that is passed an instance of the application's ECO Space. The constructor then uses this instance to initialize the form's ECO Space property. An ECO enabled form also provides fields to hold a root handle, an expression handle, and code to connect the root handle to the ECO Space.

**Note:** The default name convention for the new class and source code file is `EcoWinFormX.pas`, where X is a progressively increasing number. You can change the default name of the class in the *Source Code Editor*. The name of the source code file can be changed in the *Project Manager* window.

## To Add a New UML Package to Your Project

1. Click on File ► New ► Other to open the *New Items* dialog box.
2. Select the *ECO/Delphi for .NET Files* category.
3. Double-click the *ECO UML Package* icon.

The *ECO UML Package Wizard* generates a source code file that contains the declarations necessary for the new UML package. You can populate the UML package with new classes, and draw the relationships between them on the class diagramming surface. As you design the classes contained in a particular UML package, the IDE generates source code into the appropriate source code file. The

generated source code will be adorned with the necessary ECO attributes to support the services provided by the ECO framework.

**Note:** The default name convention for the new UML package is `CoreClassesXPackage`, where X is a progressively increasing number. You can change the default UML package name in the *Source Code Editor*. You can change the source code file name in the *Project Manager* window.

## To Create a New Subclass of the EcoSpace Class

---

1. Click on File ► New ► Other to open the *New Items* dialog box.
2. Select the *ECO/Delphi for .NET Files* category.
3. Double-click the *ECO Space* icon.

The *ECO Space Wizard* will generate a source code file that contains a new subclass of the `Borland.Eco.Handles.EcoSpace` class.

# Using the OCL Expression Editor

---

You can use the OCL Expression Editor to create the OCL expressions that define business logic for your model and the application that is derived from the model. The OCL Expression Editor is available from the Object Inspector, when you have created a new ExpressionHandle on your Windows Form.

## To open the Expression Editor

---

1. Create a new ECO application or open an existing one.
2. In the *Model View* tab, double-click on the *CoreClassesPackage* icon.  
This opens the model diagram surface.
3. Right-click to display the context menu and choose Add ➤ Class.
4. Name the class and add any attributes or operations you want.
5. From the *Project Manager* tab, select the WinForm icon.
6. From the *Tools Palette*, drag-and-drop an *ExpressionHandle* onto the Windows Form.
7. Select the root handle (rhRoot).
8. Click the *EcoSpaceType* field in the *Object Inspector* and set the value to the name of an ECO Space.
9. Select the ExpressionHandle and double-click the *Expression* field in the *Object Inspector*.  
This displays the *OCL Expression Editor*.

## To add an expression in the editor

---

1. Double-click on the name of one of the displayed classes in the right-hand pane of the dialog box.  
**Note:** A list of valid objects and expressions appears in the right-hand pane, from which you can choose to build a complex expression. These change based on the context of what you add to the textbox on the left side of the dialog. If the list does not appear, make sure that the *EcoSpaceType* property for the root handle is set to a valid ECO Space.
2. If you want to view the data types for the attributes of the class whose name you typed, select the *Show Types* checkbox.

If you are referencing subclasses with your expression handles, they will display the attributes from their abstract base class, as well as any attributes of their own.

3. Double-click an expression element from the right-hand list, to append it to the entry in the expression textbox.
4. Continue adding elements in this way, if necessary.
5. If you add an element that has tokens, for instance, for data types, you can replace those tokens with actual values in the expression textbox.
6. When you have completed the expression, click *OK*.

**Note:** As you add items to the expression, ECO automatically parses the expression to make sure it is valid. Keep in mind that the expression may be valid without being logically correct.

## **To use the OCL Expression Editor when adding columns**

---

1. With an expression handle selected, double-click the *Columns* property field in the *Object Inspector*.  
This displays the *Column Collection Editor*.
2. Click *Add* to add a new column.
3. In the properties list on the right-hand side of the dialog, click the ellipsis (...) button next to the *Expression* field.  
This displays the *OCL Expression Editor*.
4. Construct your expression by double-clicking elements from the right-hand pane, until you are satisfied with the results.
5. Click *OK*.
6. Click *OK* to close the *Column Collection Editor*.

At runtime, this adds a new column to any data grid component that is linked to the expression handle.

# Using the Overview Window

---

Sizable, real-world models will probably not fit entirely within the diagram window. To help you view the diagram, you can use the *Overview* window.

## To Scroll the Model with the Overview Window

---

1. Click the *Overview* button in the lower right corner of the diagram.

A miniature view of the entire diagram is displayed in its own sizable window. A smaller rectangle within the *Overview* window contains that portion of the model that is currently displayed on the diagram.

2. Click and drag the floating rectangle within the *Overview* window to dynamically scroll to the portion of the diagram you want to view.
3. Click anywhere outside of the *Overview* window to close it.

**Tip:** The *Overview* window has a sizing grip in the upper left corner. Click and drag the sizing grip to resize the *Overview* window. As you resize the window, its contents are scaled to fit the current size. Resizing is useful when the model is large enough that you cannot read the text on the miniature representation in the *Overview* window.

## Source Control

---

# Adding Files to the Source Control Project

---

You must first add a file to your Delphi 8 for .NET project, after the project has already been committed to the source control repository. You can then commit both the project file and your local working files to the source control repository.

## To add the active working file

---

1. Create a new file or open an existing file in Delphi 8 for .NET.

2. Choose Team ► Add Files.

This displays the Add Files dialog box, with a list of the files included in your project that can be added to the repository.

3. Select the checkboxes next to the files you want to add.

4. Click *OK*.

This displays the *Comments* dialog.

5. Write a comment. If you want to apply the same comment to all of the files, check the Apply same comment to all check box. If you leave this unchecked, the Comments dialog displays once for each file you are adding.

6. Click *OK*.

**Tip:** You can select or deselect all of the listed files at once by clicking the *Check All* or *Uncheck All* buttons



# Checking In Files

---

When you want to update the repository image with your changed files, you can do so with the *check in* operation. While this puts your file into the repository and causes the source control system to version the file, you need to commit your changes if you want to permanently update the repository image.

## To check files into the repository

---

1. Choose Team ► Check In Files....

This displays the *Check In Files* dialog box.

2. Select the check boxes next to the filenames of the files you want to check in.
3. If you want also to keep your files checked out, select the *Keep checked out* check box.
4. Click *OK*.

This displays the *Comments* dialog.

5. Write a comment. If you want to apply the same comment to all of the files, check the *Apply same comment to all* check box. If you leave this unchecked, the *Comments* dialog displays once for each file you are checking in.
6. Click *OK*.

**Tip:** You can select or deselect all of your files at once by clicking the *Check All* or *Uncheck All* buttons, respectively.

# Checking Out Files

---

When you check files out of the source control system using the Delphi 8 for .NET SCM feature, the product performs a check out operation and a synchronization at the same time.

## To check out a file

---

1. Choose Team ► Check Out Files....

This displays the *Check Out Files* dialog.

2. Select the checkboxes next to the files you want to check out.

3. Click *OK*.

4. If the check out operation encounters unsynchronized changes between files you already have on your working system and those that you are checking out, resolve the conflicts that appear in the *Synchronization* dialog box.

5. Click *OK*.

This displays the *Comments* dialog.

6. Write a comment. If you want to apply the same comment to all of the files, check the *Apply same comment to all* check box. If you leave this unchecked, the *Comments* dialog displays once for each file you are checking out.

7. Click *OK*.

**Tip:** You can select or deselect all of your files at the same time, by clicking the *Check All* or *Uncheck All* buttons, respectively.

# Using the Commit Browser

---

The Delphi 8 for .NET *Commit Browser* provides the capability to browse, select, and commit multiple files or entire branches from your project to the source control system repository or database. The *Commit Browser* provides standard options such as add, remove, check out, check in, undo checkout, history view, and version differencing.

## To commit a file

---

1. Choose the *Commits* tab to display a list of all potential commit candidate files.
2. Check the *Trim File Path* check box to limit the displayed name to the file name only.
3. For each file listed, select the *Commit* action from the *Action* drop-down list box.  
When you select an action for a file, the *Individual Comment* tab is activated.
4. Add an individual comment for the current file.

**Note:** If you want to add one comment to be applied to all of the files, use the *Summary Comment*, instead.

## To choose an action option

---

1. Choose Team ► Commit Browser.  
The available files are listed in the *Commits* tab.
2. From the *Action* drop list, choose the action you want to perform for each listed file.  
**Note:** If you do not want a file to be affected by any actions, choose the *No activity* action.
3. Click *Commit*.

## To add a summary comment

---

1. Click the *Summary Comment* tab.
2. Add your comment in the comment field.  
When you commit the files, the summary comment is added.

3. If you want the summary comment to override any existing individual comments, check the *Use Summary Comment* check box.
4. If you want to apply the comment to multiple, selected files, select the *Use Summary Comment* checkbox. By default, Delphi 8 for .NET inserts your summary comment in front of any individual comment already existing for the file.

## To add an individual comment

---

1. Select an action for each file.
2. Add a comment to the *Individual Comment* window for each selected file.
3. Check the *Use Individual Comment* check box to override any summary comments that might be added already.

When you commit the files, the summary comment is added.

## To view the local source

---

1. Click the *Local Source* tab in the lower pane of the *Commit Browser*.
2. The source code of the selected file is displayed in the lower pane.

## To view the remote source

---

1. Click the *Remote Source* tab in the lower pane of the *Commit Browser*.
2. The source code of the remote copy of the selected file is displayed in the lower pane.

## To view history

---

1. Click the *Diff. and History* tab in the lower pane of the *Commit Browser*.
2. Click *Show History...* to display a report of the history of changes made to the files. This report contains timestamps of check ins and check outs as well as comments.

## To view conflicts

---

1. Click the *Diff. and History* tab in the lower pane of the *Commit Browser*.
2. Click *Show Difference...* to display a report that lists all conflicts between the selected local and remote sources.
3. Review the conflicts that are displayed in the pane. They might be colored differently, or they might be marked with a conflict tag, depending on the source control system you are currently using.

**Note:** If there are no conflicts, the system displays a confirmation alert to that effect.

4. When you have resolved the conflicts in your files, initiate the *Commit Browser* again and recommit the files.

# Configuring Source Control Providers

---

The Delphi 8 for .NET source control functionality automatically detects your installed source control provider, assuming it has an SCC API integration component. For example, Borland StarTeam provides a separately installable StarTeam SCC Integrator, downloadable from the Borland website. If you are using one of the Windows-enabled CVS products, you might need to download an integrator, such as the Jalindi Igloo software. Once that software is installed, Delphi 8 for .NET detects its presence. You can configure the provider by supplying a valid source control system username.

## To configure the provider

---

1. Choose Tools ► Options.  
This displays the *Options* dialog.
2. Select *Source Control Manager Options* from the tree list.  
This displays the *Source Control Manager Options* page.  
If your provider is installed and includes the SCC API integration, its name appears in the *Source Code Control Providers* drop-down list box.
3. If you have multiple providers installed, choose the provider you want to use from the *Source Code Control Providers* drop-down list box.
4. Enter the valid source control system user ID in the *User Name* text box.
5. Click *OK*.

# Connecting to the Source Control Repository

---

You need to connect to a source control system repository, generally on a remote server, before you can begin managing your source files within the system. If you operate in a very small shop, you might use a repository that is installed on your local system. Like any database system, you must connect to the repository before you can view or update its contents. Whenever you attempt one of the Team operations, you are prompted by Delphi 8 for .NET to log in to the source control system, if you are not already logged in. For the purpose of describing this process, the following procedure starts with pulling a project from the repository. However, many of the Team menu commands initiate this process.

## To connect to a repository

---

1. Choose Team ► Pull Project from Source Control.

If you are not already connected to the source control system, Delphi 8 for .NET displays the connection dialog box.

2. Enter your user ID in the *Username* textbox.
3. Enter your password in the *Password* textbox.
4. Enter the repository name in the *Database* textbox, or click the *Browse...* button to locate the repository on your network.
5. Click *OK* to connect.

# Placing a Project into Source Control

---

## To place a project into source control

---

1. Choose Team ► Place Project into Source Control... to start the wizard.
2. On the first page of the wizard, choose the source control system from which you want to pull the project. If you have only one source control system, it is listed as the default.
3. Click *Next*.

## To select the source control project

---

1. On the next page of the wizard, enter the name of a valid project or click the *ellipsis(...)* button.  
If you have not already logged in to your source control system, this displays the connection dialog for your particular source control system. For more information on how to log on, see the subtask *To select a file if you are not yet logged on to the source control system*.
2. When you have located the project and it appears in the text box of this wizard page, click *Place*.  
This pulls the project and places it into your target directory. The results of the operation are displayed in the *Place project Wizard: Status page* dialog.

## To select a project if you are not yet logged on to the source control system

---

1. In the *Open Existing Project* dialog, select a server from the *Server description:* drop-down list box.
2. Click *Log On*.  
This displays the source control system log on dialog.
3. Enter your user ID in the *User name:* textbox.
4. Enter your password in the *Password:* textbox.  
If you enter the correct log on information, the *Open Existing Project* dialog is displayed with the available projects displayed.



5. Select a project from the list of those displayed in the *Project:* list box.
6. Select a module or view from the *View:* drop-down list box.

**Note:** Your system may call this a view, a module, a project, or may use some other terminology to refer to the basic project unit.

## To select a project if you are logged on to the source control system

---

1. In the *Open Existing Project* dialog, select a server from the *Server description:* drop-down list box.
2. Select a project from the list of those displayed in the *Project:* list box.
3. Select a module or view from the *View:* drop-down list box.

**Note:** Your system may call this a view, a module, a project, or may use some other terminology to refer to the basic project unit.

4. Click *OK*.

**Note:** If your system supports the notion of nested branches or folders, you might be presented with another dialog box, from which you can select the target branch.

# Pulling a Project from Source Control

---

You run the *Pull Project* wizard to retrieve a project from the source control repository into your local working directory. You can perform this task after you have connected to the source control system repository, or the wizard prompts you to connect, if it does not detect a connection.

## To pull a project

---

1. Choose Team ► Pull Project from Source Control... to start the wizard.
2. On the first page of the wizard, choose the source control system from which you want to pull the project. If you have only one source control system, it is listed as the default.
3. Click *Next*.

## To select a target directory for your project

---

1. On the next page of the wizard, enter the name of a valid target directory name on your local system or click the *ellipsis(...)* button to display the file browser and locate a valid target directory.  
  
If you enter a new directory name into the field, the wizard creates the new directory for you.
2. Click *OK* to close the file browser window.
3. Click *Next*.
4. Click *Pull* to pull the project from the repository into your target directory.  
  
The wizard displays the status of the operation in a separate *Pull project Wizard: Status page* dialog box.
5. Click *Close* to close the *Pull project Wizard: Status page*.

## To select the source control project

---

1. On the next page of the wizard, enter the name of a valid project or click the *ellipsis(...)* button.  
  
If you have not already logged in to your source control system, this displays the connection dialog for your particular source control system. For more information on how to log on, see the subtask *To select a file if you are not yet logged on to the source control system*.

2. When you have located the project and it appears in the text box of this wizard page, click *Pull*.

This pulls the project and places it into your target directory. The results of the operation are displayed in the *Pull project Wizard: Status page* dialog.

## To select a project if you are not yet logged on to the source control system

---

1. In the *Open Existing Project* dialog, select a server from the *Server description:* drop-down list box.

2. Click *Log On*.

This displays the source control system log on dialog.

3. Enter your user ID in the *User name:* textbox.

4. Enter your password in the *Password:* textbox.

If you enter the correct log on information, the *Open Existing Project* dialog is displayed with the available projects displayed.

5. Select a project from the list of those displayed in the *Project:* list box.

6. Select a module or view from the *View:* drop-down list box.

**Note:** Your system may call this a view, a module, a project, or may use some other terminology to refer to the basic project unit.

## To select a project if you are logged on to the source control system

---

1. In the *Open Existing Project* dialog, select a server from the *Server description:* drop-down list box.

2. Select a project from the list of those displayed in the *Project:* list box.

3. Select a module or view from the *View:* drop-down list box.

**Note:** Your system may call this a view, a module, a project, or may use some other terminology to refer to the basic project unit.

4. Click *OK*.

**Note:** If your system supports the notion of nested branches or folders, you might be presented with another dialog box, from which you can select the target branch.

# Removing Files from Source Control

---

When you remove files from your source control system from within Delphi 8 for .NET, you delete the selected files from the source control repository, from your local workspace, and from the Delphi 8 for .NET project.

## To remove inactive or multiple files

---

1. Choose Team ► Remove Files.

This displays the *Remove Files* dialog.

2. Select the check boxes next to the files you want to remove.
3. Click *OK*.

Delphi 8 for .NET prompts you to confirm that you want to remove the files.

4. Click *OK* to confirm the removal.

**Tip:** You can check or uncheck the entire list of files by clicking the *Check All* or *Uncheck All* buttons, respectively.

# Undoing a Check Out Operation

---

If you undo the check out of selected files, you void all changes to those files.

**Note:** You must have Check Out access rights to use this command. Check with your system administrator to find out more about your SCM privileges.

## To undo a file check out

---

1. Choose Team ► Undo Check Out Files.

This displays the *Undo Check Out* dialog, which lists the files that you checked out.

2. Select the check boxes next to the files for which you want to undo the check out.

**Tip:** You can check or uncheck all of the files at once by clicking the *Check All* or *Uncheck All* buttons, respectively.

3. Click *OK*.

**Note:** If you have left any check boxes unchecked, this does not check in the corresponding files. It just means that you won't undo the check out of those files and that you intend to retain any changes to those files. You must still perform a check in on them at some point.

# Running an SCC Application

---

In addition to performing basic source control operations from within Delphi 8 for .NET, you can run a separate instance of your source control application in its own process from within Delphi 8 for .NET.

## To run an SCC Application

---

1. Choose Team ► Run SCC App.

This initiates a session of your source control system, assuming it supports SCC API.

2. Perform source code control operations in the session instance.
3. Exit the session instance prior to shutting down Delphi 8 for .NET.





# Building VCL Forms Applications With Graphics

---

Each of the procedures listed below builds a VCL Form application that uses graphics. Build one or more of the examples. Then refer to Chapter 12, "Overview of Graphic Programming," in the Delphi 7 Developer's Guide for information on other graphics features that you can add to these basic VCL Form applications.

---

1. Draw straight lines.
2. Draw rectangles and ellipses.
3. Draw a polygon.
4. Display a bitmap image.
5. Place a bitmap in a combo box.

# Building a VCL Forms MDI Application Without Using a Wizard

---

## The basic steps to create a new MDI application with a child window without using a wizard are

---

1. Create a main window form (MDI parent window).
2. Create a child window form.
3. Have the main window create the child window under user control.
4. Write the event handler code to close the child window.
5. Create the main menu and commands.
6. Add the event handlers for the commands.
7. Compile and run the application.

## To create the main window form

---

1. Choose File ► New ► Application.  
The Janeva designer displays.
2. In the *Object Inspector*, set the *FormStyle* property to *fsMDIForm*.
3. Enter a more descriptive name such as *frMain* for the *Name* property.
4. Save the unit file with a more descriptive name, such as *uMain.pas*.

## To create a child window

---

1. Choose File ► New ► Form
2. In the *Object Inspector*, set the *FormStyle* property to *fsMDIChild*.

3. Enter a more descriptive name such as frChild for the Name property.
4. Save the unit file as uChild.pas.

## To have the main window create the child window

---

1. Choose Project ► Options  
The *Project Options* dialog displays.
2. Select frChild from *Auto-create forms*: list and click the right-angle button to move it to the *Available forms*: list and click OK.
3. Select the frMain form to activate it; then switch to the Code view.
4. Scroll to the `uses` section and add `uChild` to the list.
5. Scroll to the `private` declarations section and enter this procedure declaration:

```
procedure CreateChildForm(const childName: string);
```

6. Scroll to the `implementation` section, and enter the code below:

```
procedure TfrMain.CreateChildForm (const childName: string);  
  var Child: TfrChild;  
  begin  
    Child := TfrChild.Create(Application);  
    Child.Caption := childName;  
  end;
```

## To write the event handler code to close the child window

---

1. If necessary, activate the frMain form; then select the *Events* tab in the *Object Inspector*.
2. Double-click the OnClose event.  
The Code editor displays with the cursor in the `TfrMain.FormClose` event handler block.
3. Enter the following code:

```
Action := caFree;
```

## To create the main menu and commands

---

1. From the *Standard* page on the Components palette, place a MainMenu component on the main form.
2. Double-click the MainMenu component.  
The Menu designer (frMain.MainMenu1) displays with the first blank menu item highlighted.
3. In the *Object Inspector* on the *Properties* tab, enter mnFile for the Name property and &File for the Caption property; then press ENTER.  
In the Menu designer, File displays as the name of the first menu item.
4. In the Menu designer, select File.  
A blank command field displays in the File group. Select the blank command.
5. In the *Object Inspector*, enter mnNewChild for the Name property and &New child for the Caption property; then press ENTER.  
In the Menu designer, New child displays as the name of the first file command, and a blank command field displays just beneath New child.
6. Select the blank command.
7. In the *Object Inspector*, enter mnCloseAll for the Name property and &Close All for the Caption property; then press ENTER.  
In the Menu designer, Close All displays as the name of the second file command.

## To add event handlers for the New child and Close All commands

---

1. If necessary, open the Menu designer and select New child.
2. In the *Object Inspector*, double-click the OnClick event on the *Events* tab.  
The Code editor displays with the cursor in the `TfrMain.mnNewChildClick` event handler block.
3. At the cursor, enter the following code:

```
CreateChildForm('Child '+IntToStr(MDIChildCount+1));
```

4. In the Menu designer, select Close All.
5. In the *Object Inspector*, double-click the OnClick event on the *Events* tab.

The Code editor displays with the cursor in the `TfmMain.mnCloseAllClick` event handler block.

6. At the cursor, enter the following code:

```
for i:=0 to MDIChildCount - 1 do  
    MDIChildren[i].Close;
```

7. Just before the code block in the event handler, declare the local variable `i`.

The first two lines of the event handler code should appear as shown here when you are done:

```
procedure TfmMain.mnCloseAllClick(Sender: TObject);  
    var i: integer;
```

**Note:** The event handler minimizes the child window in the main window. To close the child window, you must add an OnClose procedure to the child form (next).

## To close the child window

---

1. Activate the child form.
2. In the *Object Inspector*, double-click the OnClose event on the *Events* tab.

The Code editor displays with the cursor in the `TfrChild.FormClose` event handler block.

3. At the cursor, enter the following statement:

```
Action := caFree;
```

## To compile and run the MDI application

---

1. Choose Run ► Run.
2. The application executes, displaying the File command.
3. Choose File ► New child one or more times.

A child window displays with each New child command.

4. Choose File  Close All.

The child windows close.

# Building a VCL Forms MDI Application Using a Wizard

---

The Janeva MDI application wizard automatically creates a project that includes the basic files for an MDI application. In addition to the Main source file, the wizard creates unit files for child and about box windows, along with the supporting forms files and resources.

## To create a new MDI application using a wizard

---

1. Choose File ► New ► Other to display the *New Items* dialog.
2. Click on the *Projects* page and double-click MDI Application.  
The *Select Directory* dialog displays.
3. Navigate to the folder in which you want to store the files for the project.
4. Click *OK*.
5. Choose Run ► Run to compile and run the application.
6. Try commands that are automatically set up by the MDI Application wizard.

# Building a VCL Forms SDI Application

---

## To create a new SDI application

---

1. Choose File ► New ► Other to display the *New Items* dialog.
2. Click on the *Projects* page and double-click SDI Application.
3. Click *OK*.
4. Choose Run ► Run to compile and run the application.



# Building a Network Application with Socket Components

---

Type introductory text only if necessary, otherwise, start the procedure. Avoid including conceptual information. For more complex procedures requiring multiple stages, use the initial taskList to outline the required steps, then use subtasks for each subsequent stage.

## To drop a component on a form

---

1. Do this. When indicating a location in the IDE, provide the locator first, followed by the action. For example, “On the *Component palette*, click the *Standard* page...”
2. Do that. Use the appropriate XML tag for IDE elements such as dialog box names and fields.
3. Do this. Save this file in the HowTo directory, using the [TBD] naming convention.
4. If the procedure exceeds nine steps, consider splitting it into subtasks.

## To drop another component on a form

---

1. Do this. Avoid more than three subtasks in a file. If documenting more than three subtasks, consider reorganizing into separate procedure files. Use a procedure task list to link procedure files together.
2. Do that.
3. Do the other.

Add optional summary information?

## To drop yet another component on a form

---

1. Do this. Avoid more than three [TBD] subtasks in a file.
2. Do that.
3. Do the other.

Add optional summary information?

**Tip:** Use a tip to provide optional information, such as a shortcut key.

**Note:** Use a note to provide important information that might prevent the procedure from working.

**Warning:** Use a warning to indicate a serious danger, such as loss of data.

# Building a VCL Forms dbExpress.NET Database Application

---

The following procedure describes how to build a dbExpress database application.

Building a VCL Forms dbExpress.NET application consists of the following major steps:

1. Set up the database connection.
2. Set up the unidirectional dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

## To add a dbExpress connection component

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. From the *General* category of the Tool palette, place a TSQLConnection component on the form.
4. Double-click the SQLConnection component to display the *Connection Editor*.
5. In the *Connection Editor*, set the *Connection Name* list to IBConnection.
6. In the *Connections Setting* box, specify the path to the InterBase database file called employee.gdb in the Database field.  
By default, the file is located in C:\Program Files\Common Files\Borland Shared\Data.
7. Accept the value in the *User\_Name* field (sysdba) and *Password* field (masterkey).
8. To test the connection, click the button with the checkmark on it (just above the *Connection Name* list).  
**Note:** By default, you are prompted to log in. Use the masterkey password.  
A message displays indicating that the connection is successful.
9. Click *OK* to close the *Connection Editor* and save your changes.

## To set up the unidirectional dataset

---

1. From the General category of the Tool Palette, place a TSQLDataSet component at the top of the form.
2. In the *Object Inspector*, select the SqlConnection property drop-down list. Set it to SqlConnection1.
3. Set the CommandText property to an SQL command, for example, Select \* from SALES.

For the SQL command, you can either type a Select statement in the Object Inspector or click the ellipsis to the right of CommandText to display the *CommandText Editor* where you can build your own query statement.

**Tip:** If you need additional help while using the *CommandText Editor*, click the *Help* button.

4. In the *Object Inspector*, set the Active property to True to open the dataset.

## To add the provider

---

1. From the *Data Access* category of the Tool palette, place a TDataSetProvider component at the top of the form.
2. In the *Object Inspector*, select the DataSet property drop-down list. Set it to SQLDataSet1.

## To add client dataset

---

1. From the *Data Access* category of the Tool palette, place a TClientDataSet component to the right of the DataSetProvider component on the form.
2. In the *Object Inspector*, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application.

A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.


## To add the data source

---

1. From the *Data Access* category of the Tool palette, place a TDataSource component to the right of the ClientDataSet on the form.
2. In the *Object Inspector*, select the DataSet property drop-down. Set it to ClientDataSet1.

## To connect a DataGrid to the DataSet

---

1. From the *Data Controls* category of the Tool palette, place a TDBGrid component on the form.
2. In the *Object Inspector*, select the DataSource property drop-down. Set the data source to DataSource1.
3. Save all files in the project; then select Run  Run.

You are prompted to enter a password. Enter masterkey.

The application compiles and displays a VCL form with a DBGrid.

# Building a VCL Forms ADO.NET Database Application

---

The following procedure describes how to build an ADO.NET database application.

Building a VCL.NET ADO.NET application consists of the following major steps:

1. Set up the database connection.
2. Set up the dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

## To add an ADO connection component

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. From the *ADO* page of the Component palette, place an ADOConnection component on the form.
4. Double-click the ADOConnection component to display the *ConnectionString* dialog.
5. If necessary, select *Use Connection String*; then click the *Build* button to display the *Link Properties* dialog.
6. On the *Provider* page of the dialog, select *Microsoft Jet 4.0 OLE DB Provider*; then click the *Next* button to display the *Connections* page.
7. On the *Connections* page, click the ellipsis button to browse for the dbdemos.mdb database. The default path to this database is C:\Program Files\Common Files\Borland Shared\Data.
8. Click *Test Connection* to confirm the connection. A dialog appears, indicating the status of the connection.
9. Click *OK* to close the *Data Link Properties* dialog. Click *OK* to close the *ConnectionString* dialog.

## To set up the dataset

---

1. From the *ADO* page, place an ADODataset component at the top of the form.

2. In the *Object Inspector*, select the Connection property drop-down list. Set it to ADOConnection1.
3. Set the CommandText property to an SQL command, for example, Select \* from orders.

You can either type the Select statement in the *Object Inspector* or click the ellipsis to the right of CommandText to display the *CommandText Editor* where you can build your own query statement.

**Tip:** If you need additional help while using the *CommandText Editor*, click the *Help* button.

4. Set the Active property to True to open the dataset.

You are prompted to log in. Use admin for the username and no password.

## To add the provider

---

1. From the *Data Access* page, place a DataSetProvider component at the top of the form.
2. In the *Object Inspector*, select the DataSet property drop-down list. Set it to ADODataset1.

## To add client dataset

---

1. From the *Data Access* page, place a ClientDataSet component to the right of the DataSetProvider component on the form.
2. In the *Object Inspector*, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application.

A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

## To add the data source

---

1. In the Component palette on the *Data Access* page, place a DataSource component to the right of the ClientDataSet on the form.

2. In the *Object Inspector*, select the DataSet property drop-down. Set it to ClientDataSet1.

## To connect a DataGrid to the DataSet

---

1. In the Component palette on the *Data Controls* page, place a DBGrid component on the form.
2. In the *Object Inspector*, select the DataSource property drop-down. Set the data source to DataSource1.
3. SelectRun ► Run.
4. You are prompted to log in. Enter admin for the username and no password.  
The application compiles and displays a VCL form with a DBGrid.



# Creating Actions in a VCL Forms Application

---

Using Delphi 8 for .NET, the following procedure illustrates how to create actions using the ActionList tool. It sets up a simple application and describes how to create an edit menu with cut and paste actions that can be used to cut and paste to a memo.

Creating the VCL application consists of the following major steps:

1. Create a main window.
2. Add main menu, actionlist, and memo tools to the form.
3. Create the cut and paste actions.
4. Add the actions to the main menu and associate with the edit action category.
5. Build and run the application.

## To create a main window

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.

## To add the main menu, actionlist, and memo to the form

---

1. Click the *Design* tab to switch to the Design view.
2. From the *Standard* category of the *Tool Palette*, add a TMainMenu, TActionList, and TMemo component to the form.

## To create the actions

---

1. Double-click ActionList1 on the form.  
The ActionList editor displays.
2. In the editor, select *New Standard Action* from the drop-down list to display the *Standard Action Classes* dialog.
3. Scroll to the TEditCut action, select it, and click *OK*.  
EditCut1 displays in the Actions list in the editor.

4. Select *New Standard Action* from the drop-down list to display the *Standard Action Classes* dialog.
5. Scroll to the *TEditPaste* action, select it, and click *OK*.  
*EditPaste1* displays in the Actions list in the editor.
6. Close the *ActionList* editor window.

## To add the cut and paste actions to the edit category in the main menu

---

1. Double-click *MainMenu1* on the form.  
The *MainMenu1* editor displays with the first blank command category selected.
2. In the *Object Inspector*, enter *Edit* for the *Caption* property and press ENTER.  
*Edit* displays as the first command category.
3. Click *Edit* to display a blank action just below it; then click the blank action to select it.
4. In the *Object Inspector*, select *EditCut1* from the drop-down list of actions.
5. Expand the list of Action properties, enter *Cut* for the *Caption* property, and enter *Edit* for the category; then press ENTER.  
*Cut* displays as the first action.
6. In the *MainMenu1* editor, click the second blank action beneath *Cut* to select it.
7. In the *Object Inspector*, select *EditPaste* from the drop-down list of actions.
8. Expand the list of Action properties if necessary, enter *Paste* for the *Caption* property, and enter *Edit* for the category; then press ENTER.  
*Paste* displays as the second action.

## To build and run the application

---

1. Save all files in the project; then choose *Run* ► *Run*.  
The application executes, displaying a form with the main menu bar and the *Edit* menu.

2. In the application, select text in the memo; then choose Edit ► Cut.

The text is cut from the memo.

3. Choose Edit ► Paste.

The text is pasted back into the memo.

# Building a VCL Forms "Hello world" Application

---

Though simple, the Windows Forms "Hello world" application demonstrates the essential steps for creating a Janeva application. The application uses a VCL Form, a control, an event, and will display a dialog in response to a user action.

Creating the "Hello world" application consists of the following steps:

1. Create a VCL Form with a button control.
2. Write the code to display "Hello world" when the button is clicked.
3. Run the application.

## To create a VCL Form

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. Click the *Design* tab to display the form view.
4. From the *Standard* page of the Tool palette, place a Button component on the form.

## To display the "Hello world" string

---

1. Select Button1 on the form.
2. In the *Object Inspector*, double-click the OnClick action on the *Events* tab.  
The Code editor displays, with the cursor in the TForm1.Button1Click event handler block.
3. Place the cursor before the `begin` reserved word; then press return.  
This creates a new line above the code block.
4. Insert the cursor on the new line created, and type the following variable declaration:  

```
var s: string;
```
5. Insert the cursor within the code block, and type the following code:

```
s:= 'Hello world!';  
ShowMessage(s);
```

## To run the "Hello world" application

---

1. Save your project files; then choose RunRun to build and run the application.

The form displays with a button called *Button1*.

2. Click *Button1*.

A dialog box displays the message "Hello World!"

3. Close the VCL form to return to the IDE.

# Using ActionManager to Create Actions in a VCL Forms Application

---

Using Delphi 8 for .NET, the following procedure illustrates how to create actions using ActionManager. It sets up a simple user interface with a text area, as would be appropriate for a text editing application, and describes how to create a file menu item with a file open action.

Building the VCL application with ActionManager actions consists of the following major steps:

1. Create a main window.
2. Add a file open action to the ActionManager.
3. Create the main menu.
4. Add the action to the menu.
5. Build and run the application.

## To create a main window

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.

## To add a file open action to ActionManager

---

1. From the *Additional* page of the Tool palette, add an ActionManager component to the form.
2. Double-click ActionManager to display the Action Manager editor.  
**Tip:** To display captions for nonvisual components such as ActionManager, choose Tools ► Environment Options. On the *Designer* tab, check *Show component captions*, and click *OK*.
3. If necessary, click the *Actions* tab.
4. Select *New Standard Action* from the drop-down list to display the *Standard Action Classes* dialog.
5. Scroll to the File category, and click the TFileOpen action.
6. Click *OK* to close the dialog.
7. In the Action Manager editor, select the File category.

*Open...* displays in the *Actions:* list box.

8. Click *Close* to close the editor.

## To create the main menu and add the File action to it

---

1. From the *Additional* page of the Tool palette, place an *ActionMainMenuBar* component on the form.
2. Open the Action Manager editor, and select the *File* category from the *Categories:* list box.
3. Drag *File* to the blank menu bar.

*File* displays on the menu bar.

## To build and run the application

---

1. Select **Run** ▶ **Run**.

The application executes, displaying a form with the main menu bar and the *File* menu.

2. Select **File** ▶ **Open**.

The Open file dialog displays.

# Building a VCL Forms Application

---

The following procedure illustrates the essential steps to building a VCL Forms application using Delphi 8 for .NET.

## To create a VCL Form

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. From the Tool palette, place components onto the form to create the user interface.
4. Write the code for the controls.

## To associate code with a control

---

1. Double-click the component on the form to which you want to apply logic. The *Code* editor displays, cursor in place within the event handler block.
2. Code your application logic.
3. Save and compile the application.



# Building an Application with XML Components

---

This example creates a Janeva application that uses an XMLDocument component to display contents in an XML file.

## The basic steps are:

---

1. Create an XML document.
2. Create a VCL form.
3. Place an XMLDocument component on the form, and associate it with the XML file.
4. Create VCL components to enable the display of XML file contents.
5. Write event handlers to display XML child node contents.
6. Compile and run the application.

## To create the XML document

---

1. Copy the text below into a file in a text editor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings [
    <!ELEMENT StockHoldings (Stock+)>
    <!ELEMENT Stock (name)>
    <!ELEMENT Stock (price)>
    <!ELEMENT Stock (symbol)>
    <!ELEMENT Stock (shares)>
]>

<StockHoldings>
  <Stock exchange="NASDAQ">
    <name>Borland</name>
    <price>10.375</price>
    <symbol>BORL</symbol>
    <shares>100</shares>
  </Stock>

  <Stock exchange="NYSE">
    <name>MyCompany</name>
    <price>8.75</price>
    <symbol>MYCO</symbol>
```

```
<shares type="preferred">25</shares>
</Stock>
</StockHoldings>
```

2. Save the file to your local drive as an XML document. Give it a name such as stock.xml.
3. Open the document in your browser.

The contents should display without error.

**Note:** In the browser, you can choose View ► Source to view the source in the text editor file.

## To create a form with an XMLDocument component

---

1. Start a new project.
2. Choose File ► New ► Other.
3. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
4. From the *Internet* page on the Component palette, place an XMLDocument component on the form.
5. In the *Object Inspector*, click the ellipse next to the FileName property, browse to the location of the XML file you created, and open it.  
The XML file is associated with the XMLDocument component.
6. In the *Object Inspector*, set the Active property to true.

## To set up the VCL components

---

1. From the *Standard* page on the Component palette, place a Memo component on the form.
2. From the *Standard* page on the Component palette, place two Button components on the form just above *Memo1*.
3. In the *Object Inspector* with *Button1* selected, enter Borland for the Caption property.
4. In the *Object Inspector* with *Button2* selected, enter MyCompany for the Caption property.

## To display child node contents in the XML file

---

1. In the *Object Inspector* with *Button1* selected, double-click the *OnClick* event on the *Events* tab.  
The Code displays with the cursor in the `TForm1.Button1Click` event handler block.
2. Enter the following code to display the stock price for the first child node when the Borland button is clicked:

```
BorlandStock:=XMLDocument1.DocumentElement.ChildNodes[0];  
Price:= BorlandStock.ChildNodes['price'].Text;  
Memo1.Text := Price;
```

3. Add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var  
    BorlandStock: IXMLNode;  
    Price: string;
```

4. In the *Object Inspector* with *Button2* selected, double-click the *OnClick* event on the *Events* tab.  
The Code displays with the cursor in the `TForm1.Button2Click` event handler block.
5. Enter the following code to display the stock price for the second child node when the MyCompany button is clicked:

```
MyCompany:=XMLDocument1.DocumentElement.ChildNodes[1];  
Price:= MyCompany.ChildNodes['price'].Text;  
Memo1.Text := Price;
```

6. Add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var  
    MyCompany: IXMLNode;  
    Price: string;
```

## To compile and run the application

---

1. Select **Run**  **Run** to compile and execute the application.

The application form displays two buttons and a memo.

2. Click the *Borland* button.

The stock price displays.

3. Click the *MyCompany* button.

The stock price displays.

# Displaying a Bitmap Image in a VCL Forms Application

---

This procedure loads a bitmap image from a file and displays it to a VCL form.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's `onClick` event handler to load and display a bitmap image.
4. Build and run the application.

## To create a VCL form and button

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. From the *Standard* category in the Tool palette, place a button component on the form.

## To provide a bitmap image

---

1. Create a directory in which to store your project files.
2. Locate a bitmap image on your local drive, and copy it to your project directory.
3. Save all files in your project to your project directory.

## To write the `OnClick` event handler

---

1. In the *Object Inspector*, double-click the Button1 `OnClick` event on the *Events* tab.  
The Code editor displays with the cursor in the `TForm1.Button1Click` event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with the name of the bitmap image in your project directory:

```
Rect := TRect.Create(0,0,100,100);  
Bitmap := TBitmap.Create;
```

```
try
    Bitmap.LoadFromFile('MyFile.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect);
finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
end;
```


**Tip:** You can change the size of the rectangle to be displayed by adjusting the [Rect](#) parameter values.

3. In the var section of the code, add these variable declarations:

```
Bitmap : TBitmap;
Rect : TRect;
```

## To run the program

---

1. Save all files in your project; then choose Run  Run.
2. Click the button to display the image bitmap in a 100 x 100-pixel rectangle in the upper left corner of the form.

# Drawing a Polygon in a VCL Forms Application

---

This procedure draws a polygon in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a polygon.
3. Build and run the application.

## To create a VCL form

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. In the form view, click the form, if necessary, to display Form1 in the *Object Inspector*.

## To write the OnPaint event handler

---

1. In the *Object Inspector*, click the *Events* tab.
2. Double-click the OnPaint event.  
The Code editor displays with the cursor in the `TForm1.FormPaint` event handler block.
3. Enter the following event handling code:

```
Canvas.Polygon ([Point(0,0), Point(0, ClientHeight),  
Point(ClientWidth, ClientHeight)]);
```

## To run the program

---

1. Save all files in your project; then select Run ► Run.
2. The applications executes, displaying a right triangle in the lower left half of the form.

# Drawing Rectangles and Ellipses in a VCL Forms Application

---

This procedure draws a rectangle and ellipse in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a rectangle and ellipse.
3. Build and run the application.

## To create a VCL form

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. Click the *Design* tab, if necessary, to display Form1 in the *Object Inspector*.

## To write the OnPaint event handler

---

1. In the *Object Inspector*, double-click the Form1 OnPaint event on the *Events* tab.  
The Code editor displays with the cursor in the `TForm1.FormPaint` event handler block.
2. Enter the following event handling code:

```
Canvas.Rectangle (0, 0, ClientWidth div 2, ClientHeight div 2);  
Canvas.Ellipse (0, 0, ClientWidth div 2, ClientHeight div 2);
```

## To run the program

---

1. Save all files in your project; then choose Run ► Run.
2. The applications executes, displaying a rectangle in the upper left quadrant, and an ellipse in the same area of the form.



# Drawing a Rounded Rectangle in a VCL Forms Application

---

This procedure draws a rounded rectangle in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a polygon.
3. Build and run the application.

## To create a VCL form

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. In the form view, click the form, if necessary, to display Form1 in the *Object Inspector*.

## To write the OnPaint event handler

---

1. In the *Object Inspector*, click the *Events* tab.
2. Double-click the OnPaint event.  
The Code editor displays with the cursor in the `TForm1.FormPaint` event handler block.
3. Enter the following event handling code:

```
Canvas.RoundRect(0, 0, ClientWidth div 2,  
ClientHeight div 2, 10, 10);
```

## To run the program

---

1. Save all files in your project; then select Run ► Run.
2. The application executes, displaying a rounded rectangle in the upper left quadrant of the form.

# Drawing Straight Lines In a VCL Forms Application

---

This procedure draws two diagonal straight lines on an image in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw the straight lines.
3. Build and run the application.

## To create a VCL form and place an image on it

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. Click the *Design* tab, if necessary, to display Form1 in the *Object Inspector*.

## To write the OnPaint event handler

---

1. In the *Object Inspector*, double-click the Form1 OnPaint event on the *Events* tab.  
The Code editor displays with the cursor in the `TForm1.FormPaint` event handler block.
2. Enter the following event handling code:

```
with Canvas do
begin
    MoveTo(0,0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
end;
```

## To run the program

---

1. Save all files in your project; then choose Run ► Run.
2. The applications executes, displaying a form with two diagonal crossing lines.  
**Tip:** To change the color of the pen to green, insert this statement following the first `MoveTo()` statement in the event handler code: `Pen.Color := clGreen;`

Experiment using other canvas and pen object properties. See "Using the properties of the Canvas object" in the Delphi 7 Developer's Guide.

# Building a Multithreaded Application

---

These are the essential steps to building a VCL Forms multithreaded application with a thread object.

---

1. Create a VCL form with a defined thread object.
2. Optionally initialize the thread.
3. Write the thread function.
4. Optionally write the cleanup code.

# Writing Cleanup Code

---

## To clean up after your thread finishes executing

---

1. Place your cleanup code in the OnTerminate event handler to centralize it.  
This ensures that all the code gets executed.
2. Do not use any thread-local variables. OnTerminate is not run as part of your thread.

**Note:** You can safely access any objects from the OnTerminate handler.

# Avoiding Simultaneous Thread Access to the Same Memory

---

Use these basic techniques to prevent other threads from accessing the same memory as your thread:

- Lock objects.
- Use critical sections.
- Use a multi-read exclusive-write synchronizer

## To lock objects

---

1. For objects such as canvas that have a `Lock` method, call the `Lock` method, as necessary, to prevent other objects from accessing the object, and call `Unlock` when locking is no longer required.
2. Call `TThreadList.LockList` to block threads from using the list object `TThreadList`, and call `TThreadList.UnlockList` when locking is no longer required.

**Note:** You can safely make calls to `TCanvas.Lock` and `TThreadList.LockList`.

## To use a critical section

---

1. Create a global instance of `TCriticalSection`.
2. Call the `Acquire` method to lock out other threads while accessing global memory.
3. Call the `Release` method so other threads can access the memory by calling `Acquire`.

The following code has a global critical section variable `LockXY` that blocks access to the global variables `X` and `Y`. To use `X` or `Y`, a thread must surround that use with calls to the critical section such as shown here:

```
LockXY.Acquire;
try
  Y := sin(X);
finally
  LockXY.Release
end;
```

**Warning:** Critical sections only work if every thread uses them to access global memory. Otherwise, problems of simultaneous access can occur.

## To use the multi-read exclusive-write synchronizer

---

1. Create a global instance of `TMultiReadExclusiveWriteSynchronizer` that is associated with the global memory you want to protect.
2. Before any thread reads from the memory, it must call `BeginRead`.
3. At the completion of reading memory, the thread must call `EndRead`.
4. Before any thread writes to the memory, it must call `BeginWrite`.
5. At the completion of writing to the memory, the thread must call `EndWrite`.

**Warning:** The multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Otherwise, problems of simultaneous access can occur.

# Defining the Thread Object

---

## To define the thread object

---

1. Choose File ► New ► Other.

The *New Items* dialog displays.

2. On the *New* page, double-click Thread Object.

The *New Thread Object* dialog displays.

3. Enter a class name, for example, TMyThread.

4. Optionally check the *Named Thread* checkbox, and enter a name for the thread, for example, MyThreadName.

**Tip:** Entering a name for Named Thread makes it easier to track the thread while debugging.

5. Press OK.

The Code Editor displays the skeleton code for the thread object.

The code generated for the new unit will look like this if you named your thread class TMyThread.

```
unit Unit1;

interface

uses
  Classes;

type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in visual components can
only be
  used in a method called using Synchronize, for example,

    Synchronize(UpdateCaption);
```



and UpdateCaption could look like,

```
procedure TMyThread.UpdateCaption;
begin
    Form1.Caption := 'Updated in a thread';
end; }

{ TMyThread }

procedure TMyThread.Execute;
begin
    { Place thread code here }
end;

end.
```

Adding a name for the thread generates additional code for the unit. It includes the Windows unit, adds the procedure `SetName`, and adds the record `TThreadNameInfo`. The name is assigned to the `ThreadNameInfo.FName` field in the record, as shown here:

```
unit Unit1;

interface

uses
    Classes {$IFDEF MSWINDOWS} , Windows {$ENDIF};

type
    TMyThread = class(TThread)
    private
        procedure SetName;
    protected
        procedure Execute; override;
    end;

implementation

{ Important: Methods and properties of objects in visual components can
only be
    used in a method called using Synchronize, for example,

        Synchronize(UpdateCaption);

and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
        Form1.Caption := 'Updated in a thread';
```

```

        end; }

{$IFDEF MSWINDOWS}
type
    TThreadNameInfo = record
        FType: LongWord;      // must be 0x1000
        FName: PChar;         // pointer to name (in user address space)
        FThreadID: LongWord;  // thread ID (-1 indicates caller thread)
        FFlags: LongWord;     // reserved for future use, must be zero
    end;
{$ENDIF}

{ TMyThread }

procedure TMyThread.SetName;
{$IFDEF MSWINDOWS}
var
    ThreadNameInfo: TThreadNameInfo;
{$ENDIF}
begin
    {$IFDEF MSWINDOWS}
        ThreadNameInfo.FType := $1000;
        ThreadNameInfo.FName := 'MyThreadName';
        ThreadNameInfo.FThreadID := $FFFFFFFF;
        ThreadNameInfo.FFlags := 0;

        try
            RaiseException( $406D1388, 0, sizeof(ThreadNameInfo) div sizeof
(LongWord), @ThreadNameInfo );
        except
            end;
    {$ENDIF}
end;

procedure TMyThread.Execute;
begin
    SetName;
    { Place thread code here }
end;

end.

```

# Handling Exceptions

---

## To handle exceptions in the thread function

---

1. Add a try...except block to the implementation of your Execute method.
2. Code the logic such as shown here:

```
procedure TMyThreadExecute;  
begin  
    try  
        while not Terminated do  
            PerformSomeTask;  
    except  
        {do something with exceptions}  
    end;  
end;
```

# Using the Main VCL Thread

---

Using the main VCL thread consists of the following basic steps:

1. Create a separate routine to handle Windows messages received by components in your application.
2. Call `CheckSynchronize` periodically.
3. Declare thread-local variables, as necessary, for exclusive use by your thread.

## To create a separate routine

---

1. Write a main thread routine that handles accessing object properties and executing object methods for all objects in your application.
2. Call the routine from your thread's `Synchronize` method.

This is an example.

```
procedure TMyThread.PushTheButton
begin
    Button1.Click;
end;
procedure TMyThread.Execute;
begin
    ...
    Synchronize(PushThebutton);
    ...
end;
```

`Synchronize` waits for the main thread to enter the message loop and then executes the passed method.

**Note:** Because `Synchronize` uses a message loop, it does not work in console applications. For console applications, use other mechanisms, such as critical sections, to protect access to VCL objects.

## To call `CheckSynchronize`

---

1. Call `CheckSynchronize` periodically within the main thread to enable background threads to synchronize execution with the main thread.

2. To ensure the safety of making background thread calls, call `CheckSynchronize` when the application is idle, for example, from an `OnIdle` event handler.

## To use a thread-local variable

---

1. Identify variables that you want to make global to all the routines running in your thread but not shared by other instances of the same thread class.
2. Declare these variables in a `threadvar` section, for example,

```
threadvar
    x: integer;
```

**Note:** Use the `threadvar` section for global variables only. Do not use it for Pointer and Function variables or types that use copy-on-write semantics, such as long strings.

# Waiting for Threads

---

## To wait for a thread to finish executing

---

1. Use the `WaitFor` method of the other thread.
2. Code your logic. For example, the following code waits for another thread to fill a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
    end;
    ThreadList1.UnlockList;
  end;
end;
```

## To wait for a task to complete

---

1. Create a `TEvent` object of global scope.
2. When a thread completes an operation other threads are waiting for, have the thread call `TEvent.SetEvent`.
3. To turn off the signal, call `TEvent.ResetEvent`.

The following example is an `OnTerminate` event handler that uses a global counter in a critical section to keep track of the number of terminating threads. When `Counter` reaches 0, the handler calls the `SetEvent` method to signal that all processes have terminated:

```
procedure TDataModule.TaskTerminateThread(Sender: TObject);
begin
  ...
  CounterGuard.Acquire; {obtain a lock on the counter}
  Dec(Counter); {decrement the global counter variable}
  if Counter = 0 then
    Event1.SetEvent; {signal if this is the last thread}
  CounterRelease; {release the lock on the counter}
  ...
end;
```

The main thread initializes `Counter`, launches the task threads, and waits for the signal that they are all done by calling the `WaitFor` method. `WaitFor` waits a specified time period for the signal to be set and returns one of the values in the table below.

The following code example shows how the main thread launches the task threads and resumes when they have completed.

```
Event1.ResetEvent; {clear the event before launching the threads}
for i := 1 to Counter do
    TaskThread.Create(False); {create and launch the task threads}
if Event1.WaitFor(20000) <> wrSignaled then
    raise Exception;
{continue with main thread}
```

**Note:** If you do not want to stop waiting for an event handler after a specified time period, pass the `WaitFor` method a parameter value of `INFINITE`. Be careful when using `INFINITE`, because your thread will hang if the anticipated signal is never received.

## To check if another thread is waiting on your thread to terminate

1. In your `Execute` procedure, implement the `Terminate` method by checking and responding to the `Terminated` property.
2. This is one way to code the logic:

```
procedure TMyThread.Execute;
begin
    while not Terminated do
        PerformSomeTask;
end;
```

### *WaitFor* return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

# Writing the Thread Function

---

The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program, because you must make sure that you do not overwrite memory that is used by other processes in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

## To implement Execute, coordinate thread execution by

---

1. Synchronizing with a main VCL thread.
2. Avoiding simultaneous access to the same memory.
3. Waiting for threads.
4. Handling exceptions.



# Placing A Bitmap Image in a Control in a VCL Forms Application

---

This procedure adds a bitmap image to a combo box in a VCL forms application.

1. Create a VCL form.
2. Place components on the form.
3. Set component properties in the Object Inspector.
4. Write event handlers for the component's drawing action.
5. Build and run the application.

## To create a VCL form with a ComboBox component

---

1. Choose File ► New ► Other.
2. In the *New Items* dialog, select Delphi for .NET Projects; then double-click Janeva Application.  
The Janeva designer displays.
3. Click the *Design* tab to display Form1.
4. From the *Win32* category of the Tool palette, place an ImageList component on the form.
5. From the *Standard* category of the Tool palette, place a ComboBox on the form.

## To set the component properties

---

1. Select ComboBox1 on the form.
2. In the *Object Inspector*, set the Style property drop-down to csOwnerDrawFixed.
3. In the *Object Inspector*, click the ellipsis next to the Items property.  
The *String List Editor* displays.
4. Enter a string you would like to associate with the bitmap image, for example, MyImage; then click *OK*.
5. Double-click ImageList1 in the form.  
The ImageList editor displays.

6. Click the *Add* button to display the *Add Images* dialog.
7. Locate a bitmap image to display in the Combo box.

To locate an image, you can search for \*.bmp images on your local drive. Select a very small image such as an icon. Copy it to your project directory, and click *Open*.

The image displays in the ImageList editor.
8. Click *OK* to close the editor.

## To add the event handler code

---


1. In the VCL form view, select ComboBox1.
2. In the Object Inspector, click the Events page, and double-click the OnDrawItem event.

The Code editor displays with cursor in the code block of the ComboBox1DrawItem event handler.
3. Enter the following code for the event handler:

```
Combobox1.Canvas.FillRect(rect);  
ImageList1.Draw(ComboBox1.Canvas, Rect.Left, Rect.Top, Index);  
Combobox1.Canvas.TextOut(Rect.Left+ImageList1.Width+2,  
    Rect.Top, ComboBox1.Items[Index]);
```

## To run the program

---

1. Save all files in your project; then choose Run  Run.

The applications executes, displaying a form with a combo box.
2. Click the combo box drop-down.

The bitmap image and the text string display as a list item.

# Creating a VCL Forms ActiveX Active Form

---

Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder, for example.

This procedure uses the VCL forms ActiveX Active Form wizard to create an Active Form containing two components. To test the control, you can deploy it to the Web. This procedure consists of the following major steps:

1. Create an ActiveX library project for an ActiveX Active Form.
2. Add controls to the Active Form.
3. Add event handling code for the controls.
4. Deploy the project to the Web.
5. Display the form and test the controls in your Web browser.

## To create an Active X library project for an ActiveX Active Form

---

1. Create a directory on your local drive for the ActiveX project. Give it an easy to find name, for example, ActiveX.
2. Create a second directory to contain the ActiveX component and an HTML file for deploying the Active Form to your Microsoft Internet Explorer Web browser. Name this directory ActiveX\_Deploy.
3. Choose FileNewOther and select the *ActiveX* page in the *New Items* dialog.
4. On the *ActiveX* page, double-click Active Form.

The *Active Form Wizard* displays.

5. Accept the default settings and click *OK*.

The wizard generates the code needed to implement the ActiveX control and adds the code to the project. If the project is already an ActiveX library, the wizard adds the control to the current project.

**Note:** If the project is not already an ActiveX library, a *Warning* dialog displays and asks you if you want to start a new ActiveX library project.

6. Click *OK* to start a new ActiveX library project.

An ActiveX Active Form displays.

## To add some functionality to the Active Form

---

1. From the *Standard* page of the Component palette, add TEdit and TButton components to the form.
2. Select the button.
3. On the *Events* tab in the *Object Inspector*, double-click the OnClick event.

The Code editor opens with the cursor in place in the `TActiveFormX.Button1Click` event handler block.

Enter the following code at the cursor:

```
ShowMessage(Edit1.text);
```

4. Save the project files to your ActiveX directory.

## To deploy the Active Form to your Web browser

---

1. Choose Project ► Web Deployment Options....

The *Web Deployment Options* dialog displays.

2. On the *Project* page, use the *Browse* button to enter the path to the ActiveX\_Deploy directory.
3. Enter the same path for the *HTML dir*.
4. For *Target URL*, enter `.\` to indicate the current directory.
5. Click *OK*.

6. Choose Project ► Web Deploy.

HTML and OCX files are created in the ActiveX\_Deploy directory.

## To test the Active Form

---

1. Launch your browser.
2. Choose File ► Open, and browse to the ActiveX\_Deploy directory.
3. Double-click the HTML file to launch it in the browser window.

The Active Form displays in the browser window.

4. Click the button.

A pop-up dialog displays the text in the Edit box.

5. Change the text, and click the button again.

The new text you entered displays in the pop-up.

# Creating a VCL Forms ActiveX Button

---

Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder, for example.

This procedure uses the VCL forms ActiveX wizard to create an ActiveX control. To test the control, you can install it on your machine as a VCL component in the IDE. To install the control, you first need to create a package for it. This procedure consists of the following major steps:

1. Create an ActiveX library project for an ActiveX button control.
2. Register the ActiveX button so its icon can be displayed in the toolbar.
3. Create a package for the ActiveX button.
4. Install the package.
5. Test the ActiveX button.

## To create an ActiveX library project for an ActiveX button control

---

1. Create a directory on your local drive for the ActiveX project. Give it an easy to find name, for example, ActiveX.
2. Choose File ► New ► Other and select the *ActiveX* page in the *New Items* dialog.
3. On the *ActiveX* page, double-click ActiveX Control.  
The *ActiveX Control Wizard* displays.
4. In the *VCL Class Name* drop-down, select *TButton*.
5. By default, *ButtonX* displays as the *New ActiveX Name*. Rename ButtonX to the name you want displayed for your ActiveX button, for example, MyActiveXButton.

**Note:** Modifications you make to the name update the Implementation Unit and Project Name. Leave the remaining fields with default values.

6. Click *OK*.

The wizard generates the code needed to implement the ActiveX control and adds the code to the project. If the project is already an ActiveX library, the wizard adds the control to the current project.

**Note:** If the project is not already an ActiveX library, a *Warning* dialog displays and asks you if you want to start a new ActiveX library project.

7. Click *OK* to start the new ActiveX Library project.

## To register the ActiveX button

---

1. Build the project and save all files to your ActiveX project directory.  
Dismiss the warning about debugging. The project builds and creates an OCX file in your project directory.
2. Choose **Run** ► **Register ActiveX Server** to register the ActiveX button.  
A dialog box displays a message indicating that registration was successful and it shows the path to the resulting OCX file.
3. Click *OK*.

## To create a new package for the ActiveX button

---

1. Choose **File** ► **New** ► **Other** to create a new package.  
The *New Items* dialog displays.
2. Double-click **Package** on the *New* page to display the *Package - package.dpk* dialog and click *Add*.
3. On the *Add unit* tab of the *Add* dialog, browse to your project directory.
4. Select the `ButtonXControl1_TLB.pas` file, and click *Open*.
5. Click *OK* to add the file to the package and return to the *Package - package.dpk* dialog.  
The *Package - package.dpk* dialog displays showing the files in the package and two required files: `rtl.dcp` and `vcl.dcp`.

## To add the required files and install the package

---

1. In the *Package - package.dpk* dialog, select `rtl.dcp`, and click *Add*.
2. On the *Add unit* tab of the *Add* dialog, browse to the `Lib` directory in Delphi, select the `rtl.dcp` file, and click *Open*; then click *OK* on the *Add* dialog.

3. In the *Package - package.dpk* dialog, select *vcl.dcp*, and click *Add*.
4. On the *Add unit* tab of the *Add* dialog, browse to the *Lib* directory in Delphi, select the *vcl.dcp* file, and click *Open*; then click *OK* on the *Add* dialog.
5. In the *Package - package.dpk* dialog, click *Compile* to compile the package.  
A dialog displays, indicating that the package has been installed. Click *OK*.
6. Click the X in the upper right corner of the *Package - package.dpk* dialog to close it.  
You are prompted to save the package.
7. Save the package to your projects directory.

## To test the button

---

1. ChooseFile ► New Application.
2. From the *ActiveX* page of the Component palette, locate your button and place it on the form.  
The button displays on the form.



# Importing .NET Controls to VCL.NET

---

There might be cases in which you want to use .NET components on your VCL.NET forms. There is no direct way to use .NET components. You can, however, wrap the components in an ActiveX wrapper, which then can be added to your VCL.NET application. Delphi 8 for .NET provides the .NETImport Wizard to accomplish this task.

## To use .NET components in a VCL.NET Form

---

1. Run the .NETImport Wizard.
2. Build the package to create an assembly file.
3. Add the assembly to the *Tool Palette*.

## To run the .NETImport Wizard

---

1. Choose File ► New ► Delphi for .NET Projects ► .NET Controls Package.

This starts the .NETImport Wizard.

2. Specify the following file:

```
\Microsoft.NET\Framework\v1.1.4322\System.Windows.Forms.dll
```

3. Click *Next*.

This displays the second page of the Wizard, and lists all of the available components.

4. Select the checkboxes next to the components you want to import.

**Note:** If you want to import all components, click the *Check All* button.

5. Click *Next*.

This displays the third page of the Wizard, which provides generation options for the units.

6. Take the defaults, and click *Next*.

This displays the fourth page of the Wizard, which allows you to set a location and a name for the package file.

7. Click *Next*.

This displays the fifth page of the Wizard, which allows you to overwrite any existing files of the same name.

8. Click *Next*.

This initiates the generation process and displays status messages for each file as it creates it, including the package (.dpk) file.

9. If you want to import additional controls, click *New*, otherwise, click *Finish*.

The package containing the units appears in the *Project Manager*.

## To build and add the package

---

1. Select the package node in the *Project Manager*.
2. Choose File ► Project ► Build <Project Name> where <Project Name> is the real name of your project.  
  
This creates the assembly file containing the package and the units.
3. Choose Components ► Installed .NET Components.
4. Click the *.NET VCL Components* tab.
5. Click *Add*.
6. Locate the package assembly, select it, and click *Open*.
7. Click *OK*.

The individual controls appear in the *Tool Palette*. You can now add the individual controls to your VCL.NET form applications.

## Reference

# Delphi Overview

---

This chapter provides a brief introduction to Delphi programs, and program organization.

# Language Overview

---

Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this language guide assume that you are using Borland development tools.

Most developers using Borland software development tools write and compile their code in the integrated development environment (IDE). Borland development tools handle many details of setting up projects and source files, such as maintenance of dependency information among units. The product also places constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, Borland development tools enforce certain file- and program-naming conventions that you can avoid if you write your programs outside of the IDE and compile them from the command prompt.

This language guide generally assumes that you are working in the IDE and that you are building applications that use the Borland Visual Component Library (VCL). Occasionally, however, Delphi-specific rules are distinguished from rules that apply to all Object Pascal programming. This text covers both the Win32 Delphi language compiler, and the Delphi for .NET language compiler. Platform-specific language differences and features are noted where necessary.

This section covers the following topics:

- Program Organization. Covers the basic language features that allow you to partition your application into units and namespaces.
- Example Programs. Small examples of both console and GUI applications are shown, with basic instructions on running the compiler from the command-line.

## Program Organization

---

Delphi programs are usually divided into source-code modules called units. Most programs begin with a program heading, which specifies a name for the program. The program heading is followed by an optional uses clause, then a block of declarations and statements. The uses clause lists units that are linked into the program; these units, which can be shared by different programs, often have uses clauses of their own.

The uses clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, most Delphi language programs do not require makefiles, header files, or preprocessor "include" directives.

## Delphi Source Files

The compiler expects to find Delphi source code in files of three kinds:

- Unit source files (which end with the .pas extension)

- Project files (which end with the .dpr extension)
- Package source files (which end with the .dpk extension)

Unit source files typically contain most of the code in an application. Each application has a single project file and several unit files; the project file, which corresponds to the program file in traditional Pascal, organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. If you use the IDE to build your application, it will produce a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called packages.

## Other Files Used to Build Applications

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically by the IDE, and include

- VCL form files (which have a .dfm extension on Win32, and .nfm on .NET)
- Resource files (which end with .res)
- Project options files (which end with .dof )

A VCL form file contains the description of the properties of the form and the components it owns. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text (a format very suitable for version control) or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to VCL form files, each project uses a resource (.res) file to hold the application's icon and other resources such as strings. By default, this file has the same name as the project (.dpr) file.

A project options (.dof) file contains compiler and linker settings, search path information, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

## Compiler-Generated Files

The first time you build an application or a package, the compiler produces a compiled unit file (.dcu on Win32, .dcuil on .NET) for each new unit used in your project; all the .dcu/.dcuil files in your project are then linked to create a single executable or shared package. The first time you build a package, the compiler produces a file for each new unit contained in the package, and then creates both a .dcp and

a package file. If you use the **GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you build a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, their .dcu/.dpu files cannot be found, you explicitly tell the compiler to reprocess them, or the interface of the unit depends on another unit which has been changed. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file and that unit has no dependencies on other units that have changed.

## Example Programs

---

The examples that follow illustrate basic features of Delphi programming. The examples show simple applications that would not normally be compiled from the IDE; you can compile them from the command line.

### A Simple Console Application

The program below is a simple console application that you can compile and run from the command prompt.

The first line declares a program called Greeting. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called MyMessage, which holds a string. (Delphi has genuine string data types.) The program then assigns the string "Hello world!" to the variable MyMessage, and sends the contents of MyMessage to the standard output using the `Writeln` procedure. (`Writeln` is defined implicitly in the `System` unit, which the compiler automatically includes in every application.)

You can type this program into a file called greeting.pas or greeting.dpr and compile it by entering

`dcc32 greeting`

to produce a Win32 executable, or

`dccil greeting`

to produce a managed .NET executable. In either case, the resulting executable prints the message Hello world!

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are most often used to write applications with graphical interfaces; hence, you would not ordinarily call `Writeln`. Moreover, the entire example program (save for `Writeln`) is in a single file. In a typical GUI application, the program heading the first line of the example would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to routines defined in unit files.

## A More Complicated Example

The next example shows a program that is divided into two files: a project file and a unit file. The project file, which you can save as `greeting.dpr`, looks like this:

The first line declares a program called `greeting`, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that the program `greeting` depends on a unit called `Unit1`. Finally, the program calls the `PrintMessage` procedure, passing to it the string `Hello World!` The `PrintMessage` procedure is defined in `Unit1`. Here is the source code for `Unit1`, which must be saved in a file called `Unit1.pas`:

`Unit1` defines a procedure called `PrintMessage` that takes a single string as an argument and sends the string to the standard output. (In Delphi, routines that do not return a value are called procedures. Routines that return a value are called functions.) Notice that `PrintMessage` is declared twice in `Unit1`. The first declaration, under the reserved word `interface`, makes `PrintMessage` available to other modules (such as `greeting`) that use `Unit1`. The second declaration, under the reserved word `implementation`, actually defines `PrintMessage`.

You can now compile `Greeting` from the command line by entering

```
dcc32 greeting
```

to produce a Win32 executable, or

```
dccil greeting
```

to produce a managed .NET executable.

There is no need to include `Unit1` as a command-line argument. When the compiler processes `greeting.dpr`, it automatically looks for unit files that the `greeting` program depends on. The resulting executable does the same thing as our first example: it prints the message `Hello world!`

## A VCL Application

Our next example is an application built using the Visual Component Library (VCL) components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of the Delphi Language. In addition to multiple units, the program uses classes and objects

The program includes a project file and two new unit files. First, the project file:

Once again, our program is called `greeting`. It uses three units: `Forms`, which is part of VCL; `Unit1`, which is associated with the application's main form (`Form1`); and `Unit2`, which is associated with another form (`Form2`).

The program makes a series of calls to an object named `Application`, which is an instance of the `TApplication` class defined in the `Forms` unit. (Every project has an automatically generated `Application` object.) Two of these calls invoke a `TApplication` method named `CreateForm`. The first



call to `CreateForm` creates `Form1`, an instance of the `TForm1` class defined in `Unit1`. The second call to `CreateForm` creates `Form2`, an instance of the `TForm2` class defined in `Unit2`.

`Unit1` looks like this:

`Unit1` creates a class named `TForm1` (derived from `TForm`) and an instance of this class, `Form1`. `TForm1` includes a `buttonButton1`, an instance of `TButton` and a procedure named `Button1Click` that is called when the user presses `Button1`. `Button1Click` hides `Form1` and it displays `Form2` (the call to `Form2.ShowModal`).

**Note:** In the previous example, `Form2.ShowModal` relies on the use of auto-created forms. While this is fine for example code, using auto-created forms is actively discouraged.

`Form2` is defined in `Unit2`:

```
unit Unit2;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
  Form2.Close;
end;

end.
```

`Unit2` creates a class named `TForm2` and an instance of this class, `Form2`. `TForm2` includes a button (`CancelButton`, an instance of `TButton`) and a label (`Label1`, an instance of `TLabel`). You can't see

this from the source code, but `Label1` displays a caption that reads Hello world! The caption is defined in `Form2`'s form file, `Unit2.dfm`.

`TForm2` declares and defines a method `CancelButtonClick` which will be invoked at runtime whenever the user presses `CancelButton`. This procedure (along with `Unit1`'s `TForm1.Button1Click`) is called an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form files for `Form1` and `Form2`.

When the `greeting` program starts, `Form1` is displayed and `Form2` is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's main form.) When the user presses the button on `Form1`, `Form2`, displays the Hello world! greeting. When the user presses the `CancelButton` or the `Close` button on the title bar, `Form2` closes.

# Programs and Units

---

This chapter provides a more detailed look at Delphi program organization.

# Programs and Units

---

A Delphi program is constructed from source code modules called units. The units are tied together by a special source code module that contains either the program, library, or package header. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Delphi 8 for .NET introduces hierarchical namespaces, giving you even more flexibility in organizing your units. Namespaces and units allow you to

- Divide large programs into modules that can be edited separately.
- Create libraries that you can share among programs.
- Distribute libraries to other developers without making the source code available.

This topic covers the overall structure of a Delphi application: the program header, unit declaration syntax, and the uses clause. Specific differences between the Win32 and .NET platforms are noted in the text. The Delphi compiler does not support .NET namespaces on the Win32 platform. The Delphi 8 for .NET compiler does support hierarchical .NET namespaces; this topic is covered in the following section, Using Namespaces with Delphi.

## Program Structure and Syntax

---

A complete, executable Delphi application consists of multiple unit modules, all tied together by a single source code module called a project file. In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use the file extension .dpr to designate the main program source module, while most other source code resides in unit files having the traditional .pas extension. To build a project, the compiler needs the project source file, and either a source file or a compiled unit file for each unit.

**Note:** Strictly speaking, you need not explicitly use any units in a project, but all programs automatically use the `System` unit and the `SysInit` unit.

The source code file for an executable Delphi application contains

- a program heading,
- a uses clause (optional), and
- a block of declarations and executable statements.

Additionally, a Delphi 8 for .NET program may contain a namespaces clause, to specify additional namespaces in which to search for generic units. This topic is covered in more detail in the section Using .NET Namespaces with Delphi.

The compiler, and hence the IDE, expect to find these three elements in a single project (.dpr) file.

## The Program Heading

The program heading specifies a name for the executable program. It consists of the reserved word `program`, followed by a valid identifier, followed by a semicolon. The identifier must match the project source file name.

The following example shows the project source file for a program called `Editor`. Since the program is called `Editor`, this project file is called `Editor.dpr`.

```
program Editor;

uses Forms, REAbout, // An "About" box
    REMain;           // Main form

{$R *.res}

begin
    Application.Title := 'Text Editor';
    Application.CreateForm(TMainForm, MainForm);
    Application.Run;
end.
```

The first line contains the program heading. The `uses` clause in this example specifies a dependency on three additional units: `Forms`, `REAbout`, and `REMain`. The `$R` compiler directive links the project's resource file into the program. Finally, the block of statements between the `begin` and `end` keywords are executed when the program runs. The project file, like all Delphi source files, ends with a period (not a semicolon).

Delphi project files are usually short, since most of a program's logic resides in its unit files. A Delphi project file typically contains only enough code to launch the application's main window, and start the event processing loop. Project files are generated and maintained automatically by the IDE, and it is seldom necessary to edit them manually.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

Borland's Delphi ignores these parameters.

In Delphi 8 for .NET, the program heading introduces its own namespace, which is called the project default namespace. This is also true for the library and package headers, when these types of projects are compiled for the .NET platform.

## The Program Uses Clause

The uses clause lists those units that are incorporated into the program. These units may in turn have uses clauses of their own. For more information on the uses clause within a unit source file, see *Unit References and the Uses Clause*, below.

The uses clause consists of the keyword `uses`, followed by a comma delimited list of units the project file directly depends on.

## The Block

The block contains a simple or structured statement that is executed when the program runs. In most program files, the block consists of a compound statement bracketed between the reserved words `begin` and `end`, whose component statements are simply method calls to the project's `Application` object. Most projects have a global `Application` variable that holds an instance of `TApplication`, `TWebApplication`, or `TServiceApplication`. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

## Unit Structure and Syntax

---

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own source (`.pas`) file.

A unit file begins with a unit heading, which is followed by the `interface` keyword. Following the `interface` keyword, the uses clause specifies a list of unit dependencies. Next comes the implementation section, followed by the optional initialization, and finalization sections. A skeleton unit source file looks like this:

```
unit Unit1;

interface

uses // List of unit dependencies goes here...

implementation

uses // List of unit dependencies goes here...

// Implementation of class methods, procedures, and functions goes here...

initialization

// Unit initialization code goes here...

finalization

// Unit finalization code goes here...
```

end.

The unit must conclude with the reserved word `end` followed by a period.

## The Unit Heading

The unit heading specifies the unit's name. It consists of the reserved word `unit`, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

would occur in a source file called `MainForm.pas`, and the file containing the compiled unit would be `MainForm.dcu` or `MainForm.dpu`.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

## The Interface Section

The interface section of a unit begins with the reserved word `interface` and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to clients. That is, to other units or programs that wish to use elements from this unit. These entities are called *public* because code in other units can access them as if they were declared in the unit itself.

The interface declaration of a procedure or function includes only the routine's signature. That is, the routine's name, parameters, and return type (for functions). The block containing executable code for the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations.

The interface declaration for a class must include declarations for all class members: fields, properties, procedures, and functions.

The interface section can include its own `uses` clause, which must appear immediately after the keyword `interface`.

## The Implementation Section

The implementation section of a unit begins with the reserved word `implementation` and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit.

That is, unlike the interface section, entities declared in the implementation section are inaccessible to other units.

The implementation section can include its own `uses` clause, which must appear immediately after the keyword `implementation`. The identifiers declared within units specified in the implementation section are only available for use within the implementation section itself. You cannot refer to such identifiers in the interface section.

## The Initialization Section

The initialization section is optional. It begins with the reserved word `initialization` and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up. So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

For units in the `interfaceuses` list, the initialization sections of the units used by a client are executed in the order in which the units appear in the client's `uses` clause.

## The Finalization Section

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word `finalization` and continues until the end of the unit. It contains statements that are executed when the main program terminates (unless the *Halt* procedure is used to terminate the program). Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initialization sections. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

**Note:** The initialization and finalization sections behave differently when code is compiled for the managed .NET environment. See the chapter on Memory Management for more information.

## Unit References and the Uses Clause

---

A `uses` clause lists units used by the program, library, or unit in which the clause appears. A `uses` clause can occur in

- the project file for a program, or library
- the interface section of a unit



- the implementation section of a unit

Most project files contain a `uses` clause, as do the interface sections of most units. The implementation section of a unit can contain its own `uses` clause as well.

The `System` unit and the `SysInit` unit are used automatically by every application and cannot be listed explicitly in the `uses` clause. (`System` implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as `SysUtils`, must be explicitly included in the `uses` clause. In most cases, all necessary units are placed in the `uses` clause by the IDE, as you add and remove units from your project.

In unit declarations and `uses` clauses, unit names must match the file names in case. In other contexts (such as qualified identifiers), unit names are case insensitive. To avoid problems with unit references, refer to the unit source file explicitly:

```
uses MyUnit in "myunit.pas";
```

If such an explicit reference appears in the project file, other source files can refer to the unit with a simple `uses` clause that does not need to match case:

```
uses Myunit;
```

## The Syntax of a Uses Clause

A `uses` clause consists of the reserved word `uses`, followed by one or more comma delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;
```

```
uses
    Forms,
    Main;
```

```
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
```

In the `uses` clause of a program or library, any unit name may be followed by the reserved word `in` and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses
    Windows, Messages, SysUtils,
    Strings in 'C:\Classes\Strings.pas', Classes;
```

Use the keyword `in` after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no

reason to do this. Using `in` is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the `in ...` construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's `uses` clause followed by `in` and a file name are considered to be part of the project; other units in the `uses` clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the *Project Manager*.

In the `uses` clause of a unit, you cannot use `in` to tell the compiler where to find a source file. Every unit must be in the compiler's search path. Moreover, unit names must match the names of their source files.

## Multiple and Indirect Unit References

The order in which units appear in the `uses` clause determines the order of their initialization and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the `uses` clause. (To access the identifier from the other unit, you would have to add a qualifier:

`UnitName.Identifier.`)

A `uses` clause need include only units used directly by the program or unit in which the clause appears. That is, if unit A references constants, types, variables, procedures, or functions that are declared in unit B, then A must use B explicitly. If B in turn references identifiers from unit C, then A is indirectly dependent on C; in this case, C needn't be included in a `uses` clause in A, but the compiler must still be able to find both B and C in order to process A.

The following example illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
// ...

unit Unit2;
interface
uses Unit1;
const b = c;
// ...

unit Unit1;
interface
```

```
const c = 1;  
// ...
```

In this example, `Prog` depends directly on `Unit2`, which depends directly on `Unit1`. Hence `Prog` is indirectly dependent on `Unit1`. Because `Unit1` does not appear in `Prog`'s `uses` clause, identifiers declared in `Unit1` are not available to `Prog`.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their `.dcl` (Win32) or `.dcuil` (.NET) files, not their source (`.pas`) files.

When a change is made in the interface section of a unit, other units that depend on the change must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

## Circular Unit References

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the `uses` clause of one interface section to the `uses` clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the `uses` clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface `uses` clauses. So the following example leads to a compilation error:

```
unit Unit1;  
interface  
uses Unit2;  
// ...  
  
unit Unit2;  
interface  
uses Unit1;  
// ...
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;  
interface  
uses Unit2;  
// ...  
  
unit Unit2;  
implementation  
uses Unit1;
```

```
interface
//...

implementation
uses Unit1;
// ...
```

To reduce the chance of circular references, it's a good idea to list units in the implementation uses clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface uses clause.

# Using Namespaces with Delphi

---

In Delphi 8 for .NET, a unit is still the basic container for types. Microsoft's Common Language Runtime (CLR) introduces another layer of organization called a namespace. In the .NET Framework, a namespace is a conceptual container of types. In Delphi 8 for .NET, a namespace is a container of Delphi units. The addition of namespaces gives Delphi the ability to access and extend classes in the .NET Framework.

Unlike traditional Delphi units, namespaces can be nested to form a containment hierarchy. Nested namespaces provide a way to organize identifiers and types, and are used to disambiguate types with the same name. Since they are a container for Delphi units, namespaces may also be used to differentiate between units of the same name, that reside in different packages.

For example, the class `MyClass` in `MyNameSpace`, is different from the class `MyClass` in `YourNameSpace`. At runtime, the CLR always refers to classes and types by their fully qualified names: the assembly name, followed by the namespace that contains the type. The CLR itself has no concept or implementation of the namespace hierarchy; it is purely a notational convenience of the programming language.

The following topics are covered:

- Project default namespaces, and namespace declaration.
- Namespace search scope.
- Using namespaces in Delphi units.

## Declaring Namespaces

---

In Delphi 8 for .NET, a project file (program, library, or package) implicitly introduces its own namespace, called the *project default namespace*. A unit may be a member of the project default namespace, or it may explicitly declare itself to be a member of a different namespace. In either case, a unit declares its namespace membership in its unit header. For example, consider the following explicit namespace declaration:

```
unit MyCompany.MyWidgets.MyUnit;
```

First, notice that namespaces are separated by dots. Namespaces do not introduce new symbols for the identifiers between the dots; the dots are part of the unit name. The source file name for this example is `MyCompany.MyWidgets.MyUnit.pas`, and the compiled output file name is `MyCompany.MyWidgets.MyUnit.dcuil`.

Second, notice that the dots imply the conceptual nesting, or containment, of one namespace within another. The example above declares the unit `MyUnit` to be a member of the `MyWidgets` namespace, which itself is contained in the `MyCompany` namespace. Again, it should be noted that this containment is for documentation purposes only.

A project default namespace declares a namespace for all of the units in the project. Consider the following declarations:

```
Program MyCompany.Programs.MyProgram;  
Library MyCompany.Libs.MyLibrary;  
Package MyCompany.Packages.MyPackage;
```

These statements establish the project default namespace for the program, library, and package, respectively. The namespace is determined by removing the rightmost identifier (and dot) from the declaration.

A unit that omits an explicit namespace is called a *generic unit*. A generic unit automatically becomes a member of the project default namespace. Given the preceding program declaration, the following unit declaration would cause the compiler to treat `MyUnit` as a member of the `MyCompany.Programs` namespace.

```
unit MyUnit;
```

The project default namespace does not affect the name of the Delphi source file for a generic unit. In the preceding example, the Delphi source file name would be `MyUnit.pas`. The compiler does however prefix the dcuil file name with the project default namespace. The resulting dcuil file in the current example would be `MyCompany.Programs.MyUnit.dcuil`.

Namespace strings are not case-sensitive. The compiler considers two namespaces that differ only in case to be equivalent. However, the compiler does preserve the case of a namespace, and will use the preserved casing in output file names, error messages, and RTTI unit identifiers. RTTI for class and type names will include the full namespace specification.

## Searching Namespaces

---

A unit must declare the other units on which it depends. As with the Win32 platform, the Delphi 8 for .NET compiler must search these units for identifiers. For units in explicit namespaces the search scope is already known, but for generic units, the compiler must establish a namespace search scope.

Consider the following unit and uses declarations:

```
unit MyCompany.Programs.Units.MyUnit1;  
uses MyCompany.Libs.Unit2, Unit3, Unit4;
```

These declarations establish `MyUnit1` as a member of the `MyCompany.Programs.Units` namespace. `MyUnit1` depends on three other units: `MyCompany.Libs.Unit2`, and the generic units, `Unit3`, and `Unit4`. The compiler can resolve identifier names in `Unit2`, since the uses clause specified the fully qualified unit name. To resolve identifier names in `Unit3` and `Unit4`, the compiler must establish a namespace search order.

## Namespace search order

Search locations can come from four possible sources: Compiler options, the project file's namespaces clause, the project default namespace, and finally, the current unit's namespace.

A project file (program, library or package) may optionally specify a list of namespaces to be searched when resolving generic unit names. The namespaces clause must appear in the project file, immediately after the program, library, or package declaration and before any other clause or block type. The namespaces clause is a list of namespace identifiers, separated by commas. A semicolon must terminate the list of namespaces.

The compiler resolves identifier names in the following order:

1. The current unit namespace (if any)
2. The project default namespace (if any)
3. Namespaces specified by the project's namespaces clause (searched in reverse order of declaration in the list).
4. Namespaces specified by compiler options.

## A namespace search example

The following example project and unit files use the namespace clause to establish a namespace search list:

```
// Project file declarations...
program MyCompany.Programs.MyProgram;
namespaces MyCompany.Libs.UIWidgets, MyCompany.Libs.Network;
```

```
// Unit source file declaration...
unit MyCompany.Programs.Units.MyUnit1;
```

Given this program example, the compiler would search namespaces in the following order:

1. `MyCompany.Programs.Units`
2. `MyCompany.Programs`
3. `MyCompany.Libs.Network`
4. `MyCompany.Libs.UIWidgets`
5. Namespaces specified by compiler options.

Note that if the current unit is generic (i.e. it does not have an explicit namespace declaration in its unit statement), then resolution begins with the project default namespace.

## Using Namespaces

---

Delphi's `uses` clause brings a module into the context of the current unit. The `uses` clause must either refer to a module by its fully qualified name (i.e. including the full namespace specification), or by its generic name, thereby relying on the namespace resolution mechanisms to locate the unit.

### Fully qualified unit names

The following example demonstrates the `uses` clause with namespaces:

```
unit MyCompany.Libs.MyUnit1
uses MyCompany.Libs.Unit2,    // Fully qualified name.
     UnitX;                   // Generic name.
```

Once a module has been brought into context, source code can refer to identifiers within that module either by the unqualified name, or by the fully qualified name (if necessary, to disambiguate identifiers with the same name in different units). The following `writeln` statements are equivalent:

```
uses MyCompany.Libs.Unit2;

begin
    writeln(MyCompany.Libs.Unit2.SomeString);
    writeln(SomeString);
end.
```

A fully qualified identifier must include the full namespace specification. In the preceding example, it would be an error to refer to `SomeString` using only a portion of the namespace:

```
writeln(Unit2.SomeString);           // ERROR!
writeln(Libs.Unit2.SomeString);      // ERROR!
writeln(MyCompany.Libs.Unit2.SomeString); // Correct.
writeln(SomeString);                 // Correct.
```

It is also an error to refer to only a portion of a namespace in the `uses` clause. There is no mechanism to import all units and symbols in a namespace. The following code does not import all units and symbols in the `MyCompany` namespace:

```
uses MyCompany;    // ERROR!
```

This restriction also applies to the `with-do` statement. The following will produce a compiler error:

```
with MyCompany.Libs do    // ERROR!
```



## Unit aliases

Namespaces can become quite long and cumbersome to type. Delphi for .NET allows you to declare a *local unit alias* to introduce an alias for a long namespace. The following uses clause introduces a local unit alias:

```
uses MyCompany.AVeryLongNamespaceDesignation.VeryDescriptiveUnitName as  
aUnit;
```

Given a unit alias, your source code can refer to an identifier by the shorter name:

```
// These two statements are equivalent.  
writeln(aUnit.SomeString);  
writeln(MyCompany.AVeryLongNamespaceDesignation.VeryDescriptiveUnitName.  
SomeString);
```

Unit aliases must not conflict with other unit or namespace identifiers. Unit aliases introduce Delphi identifiers, and therefore cannot contain dots. A unit alias is local to the unit in which it is declared. A unit alias does not obscure the fully qualified unit name that it aliases; you may always refer to an identifier either by its fully qualified name, or by a unit alias.

# Fundamental Syntactic Elements

---

This section describes the fundamental syntactic elements, or the building blocks of the Delphi language.

# Fundamental Syntactic Elements

---

This topic introduces the Delphi language character set, and describes the syntax for declaring:

- Identifiers
- Numbers
- Character strings
- Labels
- Source code comments

## The Delphi Character Set

---

The Delphi Language uses the ASCII character set, including the letters *A* through *Z* and *a* through *z*, the digits *0* through *9*, and other standard characters. It is not case-sensitive. The space character (ASCII 32) and the control characters (ASCII 0 through 31 including ASCII 13, the return or end-of-line character) are called *blanks*.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

## The Delphi Character Set and Basic Syntax

---

On the simplest level, a program is a sequence of tokens delimited by separators. A token is the smallest meaningful unit of text in a program. A separator is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;  
Price := 10;
```

Tokens are categorized as special symbols, identifiers, reserved words, directives, numerals, labels, and character strings. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

## Special Symbols

Special symbols are non-alphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols:

# \$ & ' ( ) \* + , - . / : ; < = > @ [ ] ^ { }

The following character pairs are also special symbols:

( \* ( . \* ) . ) .. // := <= >= <>

The following table shows equivalent symbols:

Special symbol	Equivalent symbols
[	(.
]	.)
{	(*
}	*)

The left bracket [ is equivalent to the character pair of left parenthesis and period (.

The right bracket ] is equivalent to the character pair of period and right parenthesis .)

The left brace { is equivalent to the character pair of left parenthesis and asterisk (\*.

The right brace } is equivalent to the character pair of right parenthesis and asterisk \*)

**Note:** %, ?, \, !, " (double quotation marks), \_ (underscore), | (pipe), and ~ (tilde) are not special characters.

## Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore (\_) and cannot contain spaces; letters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Since the Delphi Language is case-insensitive, an identifier like `CalculateValue` could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation. For more information, see the topic, *Unit References and the Uses Clause*.

## Qualified Identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to qualify the identifier. The syntax for a qualified identifier is

*identifier1.identifier2*

where *identifier1* qualifies *identifier2*. For example, if two units each declare a variable called `CurrentValue`, you can specify that you want to access the `CurrentValue` in `Unit2` by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the `Click` method in `Button1` of `Form1`.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in Blocks and scope.

## Reserved Words

The following reserved words cannot be redefined or used as identifiers.

### ***Reserved Words***

and	else	inherited	packed	then
array	end	initializationprocedure	threadvar	
as	except	inline	program	to
asm	exports	interface	property	try
begin	file	is	raise	type
case	final	label	record	unit
class	finalization	library	repeat	unsafe
const	finally	mod	resourcestringuntil	
constructor	for	nil	sealed	uses
destructor	function	not	set	var
dispinterfacegoto		object	shl	while
div	if	of	shr	with
do	implementationor		static	xor
downto	in	out	string	

In addition to the words above, `private`, `protected`, `public`, `published`, and `automated` act as reserved words within class type declarations, but are otherwise treated as directives. The words `at` and `on` also have special meanings, and should be treated as reserved words.

## Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in the Delphi language, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence -- although it is inadvisable to do so -- you can define an identifier that looks exactly like a directive.

### *Directives*

absolute	dynamic	messageprivate	resident
abstract	export	name	protected safecall
assembler	external	near	public stdcall
automated	far	nodefaultpublished	stored
cdecl	forward	overload	read varargs
contains	implementsoverride	readonly	virtual
default	index	package	register write
deprecatedlibrary	pascal	reintroduce	writeonly
dispid	local	platform	requires

## Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the + or - operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character E or e occurs within a real, it means "times ten to the power of". For example, 7E2 means  $7 * 10^2$ , and 12.25e+6 and 12.25e6 both mean  $12.25 * 10^6$ .

The dollar-sign prefix indicates a hexadecimal numeral for example, \$8F. Hexadecimal numbers without a preceding - unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the Integer (32-bit integer) where a warning is raised. In this case, values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

For more information about real and integer types, see Data Types, Variables, and Constants. For information about the data types of numerals, see True constants.

## Labels

A label is a standard Delphi language identifier with the exception that, unlike other identifiers, labels can start with a digit. Numeric labels can include no more than ten digits - that is, a numeral between 0 and 9999999999.

Labels are used in goto statements. For more information about goto statements and labels, see Goto statements.

## Character Strings

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND' { BORLAND }
'You'll see' { You'll see }
'' { ' }
'' { null string }
' ' { a space }
```

A control string is a sequence of one or more control characters, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-returnline-feed between 'Line 1' and 'Line 2'. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator or simply combine them into a single quoted string.)

A character string's length is the number of characters in the string. A character string of any length is compatible with any string type and with the PChar type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (with compiler directive {\$X+}), a

nonempty character string of length `n` is compatible with zero-based arrays and packed arrays of `n` characters. For more information, see [Datatypes](#), [Variables](#), and [Constants](#).

## Comments and Compiler Directives

---

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
(* Text between a left-parenthesis-plus-asterisk and an asterisk-
plus-right-parenthesis is also a comment *)
// Any text between a double-slash and the end of the line
constitutes a comment.
```

Comments that are alike cannot be nested. For instance, `{{}}` will not work, but `(*{ }*)` will. This is useful for commenting out sections of code that also contain comments.

A comment that contains a dollar sign (\$) immediately after the opening `{` or `(*` is a compiler directive. For example,

```
{ $WARNINGS OFF }
```

tells the compiler not to generate warning messages.



# Declarations and Statements

---

This topic describes the syntax of Delphi declarations and statements.

Aside from the `uses` clause (and reserved words like `implementation` that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

This topic covers the following items:

- Declarations
- Simple statements such as assignment
- Structured statements such as conditional tests (e.g., `if-then`, and `case`), iteration (e.g., `for`, and `while`).

## Declarations

---

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like 26057 are not identifiers.) Identifiers must be declared before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable `Result` when it occurs inside a function block, and the variable `Self` when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called `Size` that holds an Extended (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called `DoThis` that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
    Size: Extended;
    Quantity: Integer;
    Description: string
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the `uses` clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the documentation for those topics.

## Hinting Directives

The 'hint' directives `platform`, `deprecated`, and `library` may be appended to any declaration. These directives will produce warnings at compile time. Hint directives can be applied to type declarations, variable declarations, class and structure declarations, field declarations within classes or records, procedure, function and method declarations, and unit declarations.

When a hint directive appears in a unit declaration, it means that the hint applies to everything in the unit. For example, the Windows 3.1 style `OleAuto.pas` unit on Windows is completely deprecated. Any reference to that unit or any symbol in that unit will produce a deprecation message.

The `platform` hinting directive on a symbol or unit indicates that it may not exist or that the implementation may vary considerably on different platforms. The `library` hinting directive on a symbol or unit indicates that the code may not exist or the implementation may vary considerably on different library architectures.

The `platform` and `library` directives do not specify which platform or library. If your goal is writing platform-independent code, you do not need to know which platform a symbol is specific to; it is sufficient that the symbol be marked as specific to *some* platform to let you know it may cause problems for your goal of portability.

In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall deprecated

var
  VersionNumber: Real library;

type
  AppError = class(Exception)
    ...
  end platform;
```

When source code is compiled in the `{SHINTS ON}` `{$WARNINGS ON}` state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use `platform` to mark items that are specific to a particular operating environment (such as Windows or .NET), `deprecated` to indicate that an item is obsolete or supported only for backward compatibility, and `library` to flag dependencies on a particular library or component framework.

The Delphi 8 for .NET compiler also recognizes the hinting directive `experimental`. You can use this directive to designate units which are in an unstable, development state. The compiler will emit a warning when it builds an application that uses the unit.

## Statements

---

Statements define algorithmic actions within a program. Simple statements like assignments and procedure calls can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

## Simple Statements

---

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and goto jumps.

## Assignment Statements

An assignment statement has the form

*variable* := *expression*

where *variable* is any variable reference, including a variable, variable typecast, dereferenced pointer, or component of a structured variable. The *expression* is any assignment-compatible expression (within a function block, variable can be replaced with the name of the function being defined. See Procedures and functions). The := symbol is sometimes called the assignment operator.

An assignment statement replaces the current value of variable with the value of expression. For example,

```
I := 3;
```

assigns the value 3 to the variable `I`. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of `I`. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

## Procedure and Function Calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;  
Transpose(A, N, M);  
Find(Smith, William);  
Writeln('Hello world!');  
DoSomething();  
Unit1.SomeProcedure;  
TMyObject.SomeMethod(X, Y);
```

With extended syntax enabled (`{ $X+ }`), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see [Procedures and functions](#).

## Goto Statements

A goto statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

*label: statement*

Declare labels like this:

```
label label;
```

You can declare several labels at once:

```
label label1, ..., labeln;
```

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and goto statement must belong to the same block. (See [Blocks and Scope](#), below.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;  
...  
...
```

```
StartHere: Beep;  
goto StartHere;
```

creates an infinite loop that calls the [Beep](#) procedure repeatedly.

Additionally, it is not possible to jump into or out of a try-finally or try-except statement.

The goto statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;  
  var X, Y, Z, Count: Integer;  
  label FoundAnAnswer;  
begin  
  Count := SomeConstant;  
  for X := 1 to Count do  
    for Y := 1 to Count do  
      for Z := 1 to Count do  
        if ... { some condition holds on X, Y, and Z } then  
          goto FoundAnAnswer;  
  
        ... { Code to execute if no answer is found }  
      Exit;  
  
FoundAnAnswer:  
  ... { Code to execute when an answer is found }  
end;
```

Notice that we are using goto to jump out of a nested loop. Never jump into a loop or other structured statement, since this can have unpredictable effects.

## Structured Statements

---

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or with statement simply executes a sequence of constituent statements.
- A conditional statement that is an if or case statement executes at most one of its constituents, depending on specified criteria.
- Loop statements including repeat, while, and for loops execute a sequence of constituent statements repeatedly.
- A special group of statements including raise, try...except, and try...finally constructions create and handle exceptions. For information about exception generation and handling, see [Exceptions](#).

## Compound Statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words `begin` and `end`, and its constituent statements are separated by semicolons. For example:

```
begin
    Z := X;
    X := Y;
    X := Y;
end;
```

The last semicolon before `end` is optional. So we could have written this as

```
begin
    Z := X;
    X := Y;
    Y := Z
end;
```

Compound statements are essential in contexts where Delphi syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
    I := SomeConstant;
    while I > 0 do
        begin
            ...
            I := I - 1;
        end;
    end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, `begin` and `end` sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

## With Statements

A `with` statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a `with` statement is

*withobjdo*statement

or

*withobj1, ..., objndo*statement

where *obj* is an expression yielding a reference to a record, object instance, class instance, interface or class type (metaclass) instance, and statement is any simple or structured statement. Within the *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone, that is, without qualifiers.

For example, given the declarations

```
type
  TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;
```

```
var
  OrderDate: TDate;
```

you could write the following with statement.

```
with OrderDate do
  if Month = 12 then
    begin
      Month := 1;
      Year := Year + 1;
    end
  else
    Month := Month + 1;
```

you could write the following with statement.

```
if OrderDate.Month = 12 then
  begin
    OrderDate.Month := 1;
    OrderDate.Year := OrderDate.Year + 1;
  end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before statement is executed. This makes with statements efficient as well as concise. It also means that assignments to a variable within statement cannot affect the interpretation of *obj* during the current execution of the with statement.

Each variable reference or method name in a with statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the with statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year;
    ...
  end;
```

When multiple objects or records appear after with, the entire statement is treated like a series of nested with statements. Thus

*with obj1, obj2, ..., objn do statement*

is equivalent to

```
with obj1 do
  with obj2 do
    ...
    with objn do
      // statement
```

In this case, each variable reference or method name in statement is interpreted, if possible, as a member of *objn*; otherwise it is interpreted, if possible, as a member of *objn1*; and so forth. The same rule applies to interpreting the *objs* themselves, so that, for instance, if *objn* is a member of both *obj1* and *obj2*, it is interpreted as *obj2.objn*.

## If Statements

There are two forms of if statement: if...then and the if...then...else. The syntax of an if...then statement is *if expression then statement*

where *expression* returns a Boolean value. If expression is True, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I / J;
```

The syntax of an if...then...else statement is

*if expression then statement1 else statement2*

where *expression* returns a Boolean value. If expression is True, then *statement1* is executed; otherwise *statement2* is executed. For example,

```
if J = 0 then
  Exit
```



```

else
    Result := I / J;

```

The then and else clauses contain one statement each, but it can be a structured statement. For example,

```

if J <> 0 then
begin
    Result := I / J;
    Count := Count + 1;
end
else if Count = Last then
    Done := True
else
    Exit;

```

Notice that there is never a semicolon between the then clause and the word else. You can place a semicolon after an entire if statement to separate it from the next statement in its block, but the then and else clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before else (in an if statement) is a common programming error.

A special difficulty arises in connection with nested if statements. The problem arises because some if statements have else clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer else clauses than if statements, it may not seem clear which else clauses are bound to which ifs. Consider a statement of the form

```

if expression1 then if expression2 then statement1 else statement2;

```

There would appear to be two ways to parse this:

```

if expression1 then [ if expression2 then statement1 else statement2 ];

```

```

if expression1 then [ if expression2 then statement1 ] else statement2;

```

The compiler always parses in the first way. That is, in real code, the statement

```

if ... { expression1 } then
    if ... { expression2 } then
        ... { statement1 }
    else
        ... { statement2 }

```

is equivalent to

```

if ... { expression1 } then
begin

```

```

    if ... {expression2} then
        ... {statement1}
    else
        ... {statement2}
    end;
end;

```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```

if ... {expression1} then
    begin
        if ... {expression2} then
            ... {statement1}
        end
    end
else
    ... {statement2};
end

```

## Case Statements

The case statement may provide a readable alternative to deeply nested if conditionals. A case statement has the form

```

case selectorExpression of
    caseList1: statement1;
    ...
    caseListn: statementn;
end

```

where *selectorExpression* is any expression of an ordinal type (string types are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus 7, True, 4 + 5 \* 3, 'A', and Integer('A') can all be used as *caseLists*, but variables and most function calls cannot. (A few built-in functions like Hi and Lo can occur in a *caseList*. See Constant expressions.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item1, ..., itemn*, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the case statement; subranges and lists cannot overlap. A case statement can have a final else clause:

```

case selectorExpression of

```

```

    caseList1: statement1;
    ...
    caseListn: statementn;
  else
    statements;
end

```

where *statements* is a semicolon-delimited sequence of statements. When a case statement is executed, at most one of *statement1 ... statementn* is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the statement to be used. If none of the *caseLists* has the same value as *selectorExpression*, then the statements in the else clause (if there is one) are executed.

The case statement

```

case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end

```

is equivalent to the nested conditional

```

if I in [1..5] then
  Caption := 'Low';
else if I in [6..10] then
  Caption := 'High';
else if (I = 0) or (I in [10..99]) then
  Caption := 'Out of range'
else
  Caption := '';

```

Other examples of case statements

```

case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X = 3;
  Yellow, Orange, Black: X := 0;
end;

case Selection of
  Done: Form1.Close;
  Compute: calculateTotal(UnitCost, Quantity);
else

```

```
        Beep;  
end;
```

## Control Loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Delphi has three kinds of control loop: repeat statements, while statements, and for statements.

You can use the standard `Break` and `Continue` procedures to control the flow of a repeat, while, or for statement. `Break` terminates the statement in which it occurs, while `Continue` begins executing the next iteration of the sequence.

## Repeat Statements

The syntax of a **repeat** statement is

```
repeatstatement1; ...; statementn;untilexpression
```

where *expression* returns a Boolean value. (The last semicolon before `until` is optional.) The repeat statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns True, the repeat statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of repeat statements include

```
repeat  
    K := I mod J;  
    I := J;  
    J := K;  
until J = 0;  
  
repeat  
    Write('Enter a value (0..9): ');  
    Readln(I);  
until (I >= 0) and (I <= 9);
```

## While Statements

A while statement is similar to a repeat statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a while statement is

```
whileexpressiondo statement
```

where *expression* returns a Boolean value and *statement* can be a compound statement. The while statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns True, execution continues.

Examples of while statements include

```
while Data[I] <> X do I := I + 1;
```

```
while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;
```

```
while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

## For Statements

A for statement, unlike a repeat or while statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a for statement is

*for counter := initialValue to finalValue do statement*

or

*for counter := initialValue downto finalValue do statement*

where

- *counter* is a local variable (declared in the block containing the for statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with counter.
- *statement* is a simple or structured statement that does not change the value of counter.

The for statement assigns the value of *initialValue* to *counter*, then executes statement repeatedly, incrementing or decrementing *counter* after each iteration. (The for...to syntax increments *counter*, while the for...downto syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the for statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a for...to statement, or less than *finalValue* in a for...downto statement, then *statement* is never executed. After the for statement terminates (provided this was not forced by a [Break](#) or an [Exit](#) procedure), the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the for...to statement is almost, but not quite, equivalent to this while construction:

```
begin
  counter := initialValue;
  while counter <= finalValue do
    begin
      ... {statement};
      counter := Succ(counter);
    end;
  end
end
```

The difference between this construction and the for...to statement is that the while loop reevaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of for statements:

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
  for J := 1 to 10 do
    begin
      X := 0;
      for K := 1 to 10 do
        X := X + Mat1[I,K] * Mat2[K,J];
      Mat[I,J] := X;
    end;

for C := Red to Blue do Check(C);
```

## Blocks and Scope

---

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

## Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
{declarations}  
begin  
    {statements}  
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more exports clauses (see Libraries and packages).

For example, in a function declaration like

```
function UpperCase(const S: string): string;  
var  
    Ch: Char;  
    L: Integer;  
    Source, Dest: PChar;  
begin  
    ...  
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. `Ch`, `L`, `Source`, and `Dest` are local variables; their declarations apply only to the `UpperCase` function block and override, in this block only, any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

## Scope

An identifier, such as a variable or function name, can be used only within the scope of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope, especially identifiers declared in functions and procedures, are sometimes called local, while identifiers with wider scope are called global.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration section of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Programs and Units.)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See Records.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Classes and Objects.)

## Naming Conflicts

When one block encloses another, the former is called the outer block and the latter the inner block. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration takes precedence over the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called `MaxValue` in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of `MaxValue` in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a `uses` clause imposes a new scope that encloses the remaining units used and the program or unit containing the `uses` clause. The first unit in a `uses` clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope, that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the `uses` clause that does declare the identifier.

The `System` and `SysInit` units are used automatically by every program or unit. The declarations in `System`, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and bypass an inner declaration by using a qualified identifier (see Qualified Identifiers) or a `with` statement (see With Statements, above).



# Expressions

---

This topic describes syntax rules of forming Delphi expressions.

The following items are covered in this topic:

- Valid Delphi Expressions
- Operators
- Function calls
- Set constructors
- Indexes
- Typecasts

## Expressions

---

An expression is a construction that returns a value. The following table shows examples of Delphi expressions:

<code>X</code>	variable
<code>@X</code>	address of the variable X
<code>15</code>	integer constant
<code>InterestRate</code>	variable
<code>Calc(X, Y)</code>	function call
<code>X * Y</code>	product of X and Y
<code>Z / (1 - Z)</code>	quotient of Z and (1 - Z)
<code>X = 1.5</code>	Boolean
<code>C in Range1</code>	Boolean
<code>not Done</code>	negation of a Boolean
<code>['a', 'b', 'c']</code>	set
<code>Char(48)</code>	value typecast

The simplest expressions are variables and constants (described in Data types, variables, and constants). More complex expressions are built from simpler ones using operators, function calls, set constructors, indexes, and typecasts.

## Operators

Operators behave like predefined functions that are part of the the Delphi language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`, called operands, with the `+` operator; when `X`

and *Y* represent integers or reals, (*X* + *Y*) returns their sum. Operators include @, not, ^, \*, /, div, mod, and, shl, shr, as, +, -, or, xor, =, >, <, <>, <=, >=, in, and is.

The operators @, not, and ^ are unary (taking one operand). All other operators are binary (taking two operands), except that + and - can function as either a unary or binary operator. A unary operator always precedes its operand (for example, -B), except for ^, which follows its operand (for example, P^). A binary operator is placed between its operands (for example, A = 7).

Some operators behave differently depending on the type of data passed to them. For example, not performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for ^, is, and in, all operators can take operands of type Variant.

The sections that follow assume some familiarity with Delphi data types.

For information about operator precedence in complex expressions, see Operator Precedence Rules, later in this topic.

## Arithmetic Operators

Arithmetic operators, which take real or integer operands, include +, -, \*, /, div, and mod.

### Binary Arithmetic Operators

Operator	Operation	Operand Types	Result Type	Example
+	addition	integer, real	integer, real	<i>X</i> + <i>Y</i>
-	subtraction	integer, real	integer, real	<i>Result</i> - 1
*	multiplication	integer, real	integer, real	<i>P</i> * <i>InterestRate</i>
/	real division	integer, real	real	<i>X</i> / 2
div	integer division	integer	integer	<i>Total</i> div <i>UnitSize</i>
mod	remainder	integer	integer	<i>Y</i> mod 6

### Unary arithmetic operators

Operator	Operation	Operand Type	Result Type	Example
+	sign identity	integer, real	integer, real	+7
-	sign negation	integer, real	integer, real	-X

The following rules apply to arithmetic operators.

- The value of *x* / *y* is of type Extended, regardless of the types of *x* and *y*. For other arithmetic operators, the result is of type Extended whenever at least one operand is a real; otherwise, the result is of type Int64 when at least one operand is of type Int64; otherwise, the result is of type

Integer. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.

- The value of `x div y` is the value of `x / y` rounded in the direction of zero to the nearest integer.
- The mod operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x (x div y) * y`.
- A runtime error occurs when `y` is zero in an expression of the form `x / y`, `x div y`, or `x mod y`.

## Boolean Operators

The Boolean operators not, and, or, and xor take operands of any Boolean type and return a value of type Boolean.

### *Boolean Operators*

Operator	Operation	Operand Types	Result Type	Example
not	negation	Boolean	Boolean	<code>not (C in MySet)</code>
and	conjunction	Boolean	Boolean	<code>Done and (Total &gt; 0)</code>
or	disjunction	Boolean	Boolean	<code>A or B</code>
xor	exclusive disjunction	Boolean	Boolean	<code>A xor B</code>

These operations are governed by standard rules of Boolean logic. For example, an expression of the form `x and y` is True if and only if both `x` and `y` are True.

## Complete Versus Short-Circuit Boolean Evaluation

The compiler supports two modes of evaluation for the and and or operators: complete evaluation and short-circuit (partial) evaluation. Complete evaluation means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. Short-circuit evaluation means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when `A` is False, the compiler won't evaluate `B`; it knows that the entire expression is False as soon as it evaluates `A`.

Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string `S`, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
  begin
    ...
```

```

        Inc(I);
    end;

```

In the case where `S` has no commas, the last iteration increments `I` to a value which is greater than the length of `S`. When the while condition is next tested, complete evaluation results in an attempt to read `S[I]`, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the while condition `(S[I] <> ',')` is not evaluated after the first part fails.

Use the `$B` compiler directive to control evaluation mode. The default state is `{ $B }`, which enables short-circuit evaluation. To enable complete evaluation locally, add the `{ $B+ }` directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting *Complete Boolean Evaluation* in the *Compiler Options* dialog (all source units will need to be recompiled).

**Note:** If either operand involves a Variant, the compiler always performs complete evaluation (even in the `{ $B }` state).

## Logical (Bitwise) Operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in `X` (in binary) is `001101` and the value stored in `Y` is `100001`, the statement

```
Z := X or Y;
```

assigns the value `101101` to `Z`.

### Logical (Bitwise) Operators

Operator	Operation	Operand Types	Result Type	Example
not	bitwise negation	integer	integer	<code>not X</code>
and	bitwise and	integer	integer	<code>X and Y</code>
or	bitwise or	integer	integer	<code>X or Y</code>
xor	bitwise xor	integer	integer	<code>X xor Y</code>
shl	bitwise shift left	integer	integer	<code>X shl 2</code>
shr	bitwise shift right	integer	integer	<code>Y shr I</code>

The following rules apply to bitwise operators.

- The result of a not operation is of the same type as the operand.
- If the operands of an and, or, or xor operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations `x shl y` and `x shr y` shift the value of `x` to the left or right by `y` bits, which (if `x` is an unsigned integer) is equivalent to multiplying or dividing `x` by  $2^y$ ; the result is of the same type as `x`. For example, if `N` stores the value `01101` (decimal 13), then `N sh 1` returns `11010` (decimal 26). Note that the value of `y` is interpreted modulo the size of the type of `x`. Thus for

example, if `x` is an integer, `x shl 40` is interpreted as `x shl 8` because an integer is 32 bits and  $40 \bmod 32$  is 8.

## String Operators

The relational operators `=`, `<>`, `<`, `>`, `<=`, and `>=` all take string operands (see Relational operators). The `+` operator concatenates two strings.

### String Operators

Operator	Operation	Operand Types	Result Type	Example
<code>+</code>	concatenation	string, packed string, characterstring		<code>S + '. '</code>

The following rules apply to string concatenation.

- The operands for `+` can be strings, packed strings (packed arrays of type `Char`), or characters. However, if one operand is of type `WideChar`, the other operand must be a long string (`AnsiString` or `WideString`).
- The result of a `+` operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

## Pointer Operators

The relational operators `<`, `>`, `<=`, and `>=` can take operands of type `PChar` and `PWideChar` (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and pointer types.

### Character-pointer operators

Operator	Operation	Operand Types	Result Type	Example
<code>+</code>	pointer addition	character pointer, integer	character pointer	<code>P + I</code>
<code>-</code>	pointer subtraction	character pointer, integer	character pointer, integer	<code>P - Q</code>
<code>^</code>	pointer dereference	pointer	base type of pointer	<code>P^</code>
<code>=</code>	equality	pointer	Boolean	<code>P = Q</code>
<code>&lt;&gt;</code>	inequality	pointer	Boolean	<code>P &lt;&gt; Q</code>

The `^` operator dereferences a pointer. Its operand can be a pointer of any type except the generic `Pointer`, which must be typecast before dereferencing.

`P = Q` is True just in case `P` and `Q` point to the same address; otherwise, `P <> Q` is True.

You can use the `+` and `-` operators to increment and decrement the offset of a character pointer. You can also use `-` to calculate the difference between the offsets of two character pointers. The following rules apply.

- If `I` is an integer and `P` is a character pointer, then `P + I` adds `I` to the address given by `P`; that is, it returns a pointer to the address `I` characters after `P`. (The expression `I + P` is equivalent to `P + I`.) `P - I` subtracts `I` from the address given by `P`; that is, it returns a pointer to the address `I` characters before `P`. This is true for `PChar` pointers; for `PWideChar` pointers `P + I` adds `SizeOf(WideChar)` to `P`.
- If `P` and `Q` are both character pointers, then `P - Q` computes the difference between the address given by `P` (the higher address) and the address given by `Q` (the lower address); that is, it returns an integer denoting the number of characters between `P` and `Q`. `P + Q` is not defined.

## Set Operators

The following operators take sets as operands.

### Set Operators

Operator	Operation	Operand Types	Result Type	Example
<code>+</code>	union	set	set	<code>Set1 + Set2</code>
<code>-</code>	difference	set	set	<code>S - T</code>
<code>*</code>	intersection	set	set	<code>S * T</code>
<code>&lt;=</code>	subset	set	Boolean	<code>Q &lt;= MySet</code>
<code>&gt;=</code>	superset	set	Boolean	<code>S1 &gt;= S2</code>
<code>=</code>	equality	set	Boolean	<code>S2 = MySet</code>
<code>&lt;&gt;</code>	inequality	set	Boolean	<code>MySet &lt;&gt; S1</code>
<code>in</code>	membership	ordinal, set	Boolean	<code>A in Set1</code>

The following rules apply to `+`, `-`, and `*`.

- An ordinal `O` is in `X + Y` if and only if `O` is in `X` or `Y` (or both). `O` is in `X - Y` if and only if `O` is in `X` but not in `Y`. `O` is in `X * Y` if and only if `O` is in both `X` and `Y`.
- The result of a `+`, `-`, or `*` operation is of the type `set of A..B`, where `A` is the smallest ordinal value in the result set and `B` is the largest.

The following rules apply to `<=`, `>=`, `=`, `<>`, and `in`.

- `X <= Y` is True just in case every member of `X` is a member of `Y`; `Z >= W` is equivalent to `W <= Z`. `U = V` is True just in case `U` and `V` contain exactly the same members; otherwise, `U <> V` is True.

- For an ordinal `O` and a set `S`, `O in S` is True just in case `O` is a member of `S`.

## Relational Operators

Relational operators are used to compare two operands. The operators `=`, `<>`, `<=`, and `>=` also apply to sets.

### Relational Operators

Operator	Operation	Operand Types	Result Type	Example
<code>=</code>	equality	simple, class, class reference, interface, string, packed string	Boolean	<code>I = Max</code>
<code>&lt;&gt;</code>	inequality	simple, class, class reference, interface, string, packed string	Boolean	<code>X &lt;&gt; Y</code>
<code>&lt;</code>	less-than	simple, string, packed string, PChar	Boolean	<code>X &lt; Y</code>
<code>&gt;</code>	greater-than	simple, string, packed string, PChar	Boolean	<code>Len &gt; 0</code>
<code>&lt;=</code>	less-than-or-equal-to	simple, string, packed string, PChar	Boolean	<code>Cnt &lt;= I</code>
<code>&gt;=</code>	greater-than-or-equal-to	simple, string, packed string, PChar	Boolean	<code>I &gt;= 1</code>

For most simple types, comparison is straightforward. For example, `I = J` is True just in case `I` and `J` have the same value, and `I <> J` is True otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordinal values that make up the characters that make up the string. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with `n` components is compared to a string, the packed string is treated as a string of length `n`.
- Use the operators `<`, `>`, `<=`, and `>=` to compare PChar (and PWideChar) operands only if the two pointers point within the same character array.
- The operators `=` and `<>` can take operands of class and class-reference types. With operands of a class type, `=` and `<>` are evaluated according the rules that apply to pointers: `C = D` is True just in case `C` and `D` point to the same instance object, and `C <> D` is True otherwise. With operands of a class-reference type, `C = D` is True just in case `C` and `D` denote the same class, and `C <> D` is True otherwise. This does not compare the data stored in the classes. For more information about classes, see [Classes and objects](#).

## Class Operators

The operators `as` and `is` take classes and instance objects as operands; `as` operates on interfaces as well. For more information, see [Classes and objects](#) and [Object interfaces](#).

The relational operators = and <> also operate on classes.

## The @ Operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see Pointers and pointer types. The following rules apply to @.

- If *X* is a variable, @*X* returns the address of *X*. (Special rules apply when *X* is a procedural variable; see Procedural types in statements and expressions.) The type of @*X* is Pointer if the default { \$T } compiler directive is in effect. In the { \$T+ } state, @*X* is of type ^*T*, where *T* is the type of *X* (this distinction is important for assignment compatibility, see Assignment-compatibility).
- If *F* is a routine (a function or procedure), @*F* returns *F*'s entry point. The type of @*F* is always Pointer.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

```
@TMyClass.DoSomething
```

points to the `DoSomething` method of `TMyClass`. For more information about classes and methods, see Classes and objects.

**Note:** When using the @ operator, it is not possible to take the address of an interface method as the address is not known at compile time and cannot be extracted at runtime.

## Operator Precedence

In complex expressions, rules of precedence determine the order in which operations are performed.

### Precedence of operators

Operators	Precedence
@, not	first (highest)
*, /, div, mod, and, shl, shr, assecond	
+, -, or, xor	third
=, <>, <, >, <=, >=, in, is	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

`X + Y * Z`

multiplies *Y* times *Z*, then adds *X* to the result; \* is performed first, because it has a higher precedence than +. But

`X - Y + Z`



first subtracts `Y` from `X`, then adds `Z` to the result; `-` and `+` have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

```
(X + Y) * Z
```

multiplies `Z` times the sum of `X` and `Y`.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

```
X = Y or X = Z
```

The intended interpretation of this is obviously

```
(X = Y) or (X = Z)
```

Without parentheses, however, the compiler follows operator precedence rules and reads it as

```
(X = (Y or X)) = Z
```

which results in a compilation error unless `Z` is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example could be written as

```
X + (Y * Z)
```

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

## Function Calls

---

Because functions return a value, function calls are expressions. For example, if you've defined a function called `Calc` that takes two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If `I` and `J` are integer variables, then `I + Calc(J, 8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I, J);
```

For more information about functions, see [Procedures and functions](#).

## Set Constructors

---

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

```
[ item1, ..., itemn ]
```

where each item is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (..) in between. When an item has the form `x..y`, it is shorthand for all the ordinals in the range from `x` to `y`, including `y`; but if `x` is greater than `y`, then `x..y`, the set `[x..y]`, denotes nothing and is the empty set. The set constructor `[ ]` denotes the empty set, while `[x]` denotes the set whose only member is the value of `x`.

Examples of set constructors:

```
[red, green, MyColor]  
[1, 5, 10..K mod 12, 23]  
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see [Sets](#).

## Indexes

---

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if `FileName` is a string variable, the expression `FileName[3]` returns the third character in the string denoted by `FileName`, while `FileName[I + 1]` returns the character immediately after the one indexed by `I`. For information about strings, see [String types](#). For information about arrays and array properties, see [Arrays and Array properties](#).

## Typecasts

---

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character `A` as an integer.

The syntax for a typecast is

```
typeIdentifier(expression)
```

If the expression is a variable, the result is called a variable typecast; otherwise, the result is a value typecast. While their syntax is the same, different rules apply to the two kinds of typecast.

## Value Typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal or pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`, which is 65, to the variable `I`.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

## Variable Typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like `Int` and `Trunc`.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
...
Shortint(MyChar) := 122;
```

assigns the character `z` (ASCII 122) to `MyChar`.

You can cast variables to a procedural type. For example, given the declarations

```

type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;

```

you can make the following assignments.

```

F := Func(P);      { Assign procedural value in P to F }
Func(P) := F;      { Assign procedural value in F to P }
@F := P;           { Assign pointer value in P to F }
P := @F;           { Assign pointer value in F to P }
N := F(N);         { Call function via F }
N := Func(P)(N);   { Call function via P }

```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```

type
  TByteRec = record
    Lo, Hi: Byte;
  end;

  TWordRec = record
    Low, High: Word;
  end;

var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;

begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $1234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;

```

In this example, `TByteRec` is used to access the low- and high-order bytes of a word, and `TWordRec` to access the low- and high-order words of a long integer. You could call the predefined functions `Lo` and `Hi` for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see [Pointers and pointer types](#). For information about casting class and interface types, see [The as operator](#) and [Interface typecasts](#).

# Data Types, Variables, and Constants

---

This section describes the fundamental data types of the Delphi language.

# Data Types, Variables, and Constants

---

This topic presents a high-level overview of Delphi data types.

## About Types

---

A type is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

The Delphi language is a 'strongly typed' language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include typecasting, pointers, Variants, Variant parts in records, and absolute addressing of variables.

There are several ways to categorize Delphi data types:

- Some types are predefined (or built-in); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either fundamental or generic. The range and format of a fundamental type is the same in all implementations of the Delphi language, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It's a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage format from one implementation of a generic type to the next could cause compatibility problems - for example, if you are streaming content to a file as raw, binary data, without type and versioning information.
- Types can be classified as simple, string, structured, pointer, procedural, or variant. In addition, type identifiers themselves can be regarded as belonging to a special 'type' because they can be passed as parameters to certain functions (such as High, Low, and SizeOf).

The outline below shows the taxonomy of Delphi data types.

```
simple
    ordinal
        integer
        character
        Boolean
        enumerated
        subrange
    real
```

```
string
structured
set
array
record
file
class
class reference
interface
pointer
procedural
Variant
type identifier
```

The standard function `SizeOf` operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) required to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a Longint variable uses four bytes of memory.

Type declarations are illustrated in the topics that follow. For general information about type declarations, see [Declaring types](#).



# Simple Types

---

Simple types - which include ordinal types and real types - define ordered sets of values.

The ordinal types covered in this topic are:

- Integer types
- Character types
- Boolean types
- Enumerated types
- Real (floating point) types

## Ordinal Types

---

Ordinal types include integer, character, Boolean, enumerated, and subrange types. An ordinal type defines an ordered set of values in which each value except the first has a unique predecessor and each value except the last has a unique successor. Further, each value has an ordinality which determines the ordering of the type. In most cases, if a value has ordinality  $n$ , its predecessor has ordinality  $n-1$  and its successor has ordinality  $n+1$ .

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
<code>Ord</code>	ordinal expression	ordinality of expression's value	Does not take Int64 arguments.
<code>Pred</code>	ordinal expression	predecessor of expression's value	
<code>Succ</code>	ordinal expression	successor of expression's value	
<code>High</code>	ordinal type identifier or variable of highest value in type ordinal type		Also operates on short-string types and arrays.
<code>Low</code>	ordinal type identifier or variable of lowest value in type ordinal type		Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type Byte is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures `Inc` and `Dec` increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if `I` is an integer variable, to `I := I + 1`.

## Integer Types

An integer type represents a subset of the whole numbers. The generic integer types are `Integer` and `Cardinal`; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the Delphi compiler.

### *Generic integer types*

Type	Range	Format	.NET Type Mapping
Integer	-2147483648..2147483647	signed 32-bit	Int32
Cardinal	0..4294967295	unsigned 32-bit	UInt32

Fundamental integer types include `Shortint`, `Smallint`, `Longint`, `Int64`, `Byte`, `Word`, and `Longword`.

### *Fundamental integer types*

Type	Range	Format	.NET Type Mapping
Shortint	-128..127	signed 8-bit	SByte
Smallint	-32768..32767	signed 16-bit	Int16
Longint	-2147483648..2147483647	signed 32-bit	Int32
Int64	-2 <sup>63</sup> ..2 <sup>63</sup> -1	signed 64-bit	Int64
Byte	0..255	unsigned 8-bit	Byte
Word	0..65535	unsigned 16-bit	UInt16
Longword	0..4294967295	unsigned 32-bit	UInt32

In general, arithmetic operations on integers return a value of type `Integer`, which is equivalent to the 32-bit `Longint`. Operations return a value of type `Int64` only when performed on one or more `Int64` operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
  ...

  I := High(Integer);
  J := I + 1;
```

To get an `Int64` return value in this situation, cast `I` as `Int64`:

```
...  
J := Int64(I) + 1;
```

For more information, see Arithmetic operators.

**Note:** Some standard routines that take integer arguments truncate Int64 values to 32 bits. However, the [High](#), [Low](#), [Succ](#), [Pred](#), [Inc](#), [Dec](#), [IntToStr](#), and [IntToHex](#) routines fully support Int64 arguments. Also, the [Round](#), [Trunc](#), [StrToInt64](#), and [StrToInt64Def](#) functions return Int64 values. A few routines cannot take Int64 values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the Shortint type has the range 128..127; hence, after execution of the code

```
var I: Shortint;  
...  
I := High(Shortint);  
I := I + 1;
```

the value of `I` is 128. If compiler range-checking is enabled, however, this code generates a runtime error.

## Character Types

The fundamental character types are AnsiChar and WideChar. AnsiChar values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. AnsiChar was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

WideChar characters use more than one byte to represent every character. In the current implementations, WideChar is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is Char, which is equivalent to AnsiChar on Win32, and to Char on the .NET platform. Because the implementation of Char is subject to change, it's a good idea to use the standard function [SizeOf](#) rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

**Note:** The WideChar type also maps to Char on the .NET platform.

A string constant of length 1, such as 'A', can denote a character value. The predefined function [Chr](#) returns the character value for any integer in the range of AnsiChar or WideChar; for example, [Chr](#) (65) returns the letter A.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```

var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;

```

`Letter` has the value A (ASCII 65).

## Boolean Types

The four predefined Boolean types are `Boolean`, `ByteBool`, `WordBool`, and `LongBool`. `Boolean` is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A `Boolean` variable occupies one byte of memory, a `ByteBool` variable also occupies one byte, a `WordBool` variable occupies two bytes (one word), and a `LongBool` variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants `True` and `False`. The following relationships hold.

<b>Boolean</b>	<b>ByteBool, WordBool, LongBool</b>
<i>False</i> < <i>True</i>	<i>False</i> <> <i>True</i>
<i>Ord(False)</i> = 0	<i>Ord(False)</i> = 0
<i>Ord(True)</i> = 1	<i>Ord(True)</i> <> 0
<i>Succ(False)</i> = <i>True</i>	<i>Succ(False)</i> = <i>True</i>
<i>Pred(True)</i> = <i>False</i>	<i>Pred(False)</i> = <i>True</i>

A value of type `ByteBool`, `LongBool`, or `WordBool` is considered `True` when its ordinality is nonzero. If such a value appears in a context where a `Boolean` is expected, the compiler automatically converts any value of nonzero ordinality to `True`.

The previous remarks refer to the ordinality of Boolean values, not to the values themselves. In Delphi, Boolean expressions cannot be equated with integers or reals. Hence, if `X` is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```

if X <> 0 then ...; { use longer expression that returns Boolean value }

var OK: Boolean;
...
if X <> 0 then OK := True;
if OK then ...;

```

## Enumerated Types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

```
type typeName = ( val1 , ... , valn )
```

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called `Suit` whose possible values are `Club`, `Diamond`, `Heart`, and `Spade`, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each *val* to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock)
```

Unfortunately, `Click` is also the name of a method defined for `TControl` and all of the objects in VCL that descend from it. So if you're writing an application and you create an event handler like

```

procedure TForm1.DBGridEnter(Sender: TObject);
var Thing: TSound;
begin
    ...
    Thing := Click;
end;

```

you'll get a compilation error; the compiler interprets `Click` within the scope of the procedure as a reference to `TForm`'s `Click` method. You can work around this by qualifying the identifier; thus, if `TSound` is declared in `MyUnit`, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
type
```

```

TSound = (tsClick, tsClack, tsClock);
TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
Answer = (ansYes, ansNo, ansMaybe)

```

You can use the (*val1*, ..., *valn*) construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare `MyCard` this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```

var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);

```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```

type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;

```

## Enumerated Types with Explicitly Assigned Ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called `Size` whose possible values include `Small`, `Medium`, and `Large`, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration. In the previous example, the `Size` type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as `Pred`, `Succ`, `Inc`, and `Dec`. In the following example, "anonymous" values in the range of `Size` are assigned to the variable `x`.

```
var x: Size;
```

```

X := Small;      // Ord(X) = 5
Y := Size(6);    // Ord(X) = 6
Inc(X);          // Ord(X) = 7

```

Any value that isn't explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn't assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

`SomeEnum` has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because `e2` and `e3` have the same ordinality, they represent the same value.

Enumerated constants without a specific value have RTTI:

```
type SomeEnum = (e1, e2, e3);
```

whereas enumerated constants with a specific value, such as the following, do not have RTTI:

```
type SomeEnum = (e1 = 1, e2 = 2, e3 = 3);
```

## Subrange Types

A subrange type represents a subset of the values in another ordinal type (called the base type). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here `TMyColors` includes the values `Green`, `Yellow`, `Orange`, `Purple`, and `White`.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```

type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';

```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *LowerBound..UpperBound* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example, if `Color` is a variable that holds the value `Green`, `Ord(Color)` returns 2 regardless of whether `Color` is of type `TColors` or `TMyColors`.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type Percentile = 0..99;
var I: Percentile;
...
I := 100;
```

produces an error,

```
...
I := 99;
Inc(I);
```

assigns the value 100 to `I` (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after `=` is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;

type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

## Real Types

---

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types on the Win32 platform.

### ***Fundamental Win32 real types***



Type	Range	Significant digits	Size in bytes
Real48	$-2.9 \times 10^{39} \dots 1.7 \times 10^{38}$	1112	6
Single	$-1.5 \times 10^{45} \dots 3.4 \times 10^{38}$	78	4
Double	$-5.0 \times 10^{324} \dots 1.7 \times 10^{308}$	1516	8
Extended	$-3.6 \times 10^{4951} \dots 1.1 \times 10^{4932}$	1920	10
Comp	$-2^{63+1} \dots 2^{63-1}$	1920	8
Currency	-922337203685477.5808.. 922337203685477.58071920		8

The following table shows how the fundamental real types map to .NET framework types.

#### ***Fundamental .NET real type mappings***

Type	.NET Mapping
Real48	Deprecated
Single	Single
Double	Double
ExtendedDouble	
Comp	Deprecated
Currency	Re-implemented as a value type using the Decimal type from the .NET Framework

The generic type Real, in its current implementation, is equivalent to Double (which maps to Double on .NET).

#### ***Generic real types***

Type	Range	Significant digits	Size in bytes
Real	$-5.0 \times 10^{324} \dots 1.7 \times 10^{308}$	1516	8

**Note:** The six-byte Real48 type was called Real in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte Real type in Delphi, you may want to change it to Real48. You can also use the `{$REALCOMPATIBILITY ON}` compiler directive to turn Real back into the six-byte type.

The following remarks apply to fundamental real types.

- Real48 is maintained for backward compatibility. Since its storage format is not native to the Intel processor architecture, it results in slower performance than other floating-point types. The Real48 type has been deprecated on the .NET platform.
- Extended offers greater precision than other real types but is less portable. Be careful using Extended if you are creating data files to share across platforms.

- The Comp (computational) type is native to the Intel processor architecture and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a Comp value.) Comp is maintained for backward compatibility only. Use the Int64 type for better performance.
- Currency is a fixed-point data type that minimizes rounding errors in monetary calculations. On the Win32 platform, it is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, Currency values are automatically divided or multiplied by 10000.

# String Types

---

This topic describes the string data types available in the Delphi language. The following types are covered:

- Short strings.
- Long strings.
- Wide (Unicode) strings.

## About String Types

---

A string represents a sequence of characters. Delphi supports the following predefined string types.

### *String types*

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	$\sim 2^{31}$ characters	4 bytes to 2GB	8-bit (ANSI) characters, DBCS ANSI, MBCS ANSI, etc.
WideString	$\sim 2^{30}$ characters	4 bytes to 2GB	Unicode characters; multi-user servers and multi-language applications

On the Win32 platform, AnsiString, sometimes called the long string, is the preferred type for most purposes. WideString is the preferred string type on the .NET platform.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as var and out parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type.

The reserved word string functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable `S` that holds a string. On the Win32 platform, the compiler interprets string (when it appears without a bracketed number after it) as AnsiString. On the .NET platform, the string type maps to the String class. You can use single byte character strings on the .NET platform, but you must explicitly declare them to be of type AnsiString.

On the Win32 platform, you can use the `{ $H- }` directive to turn string into ShortString. The `{ $H- }` directive is deprecated on the .NET platform.

The standard function `Length` returns the number of characters in a string. The `SetLength` procedure adjusts the length of a string.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the

shorter string takes on a greater-than value. For example, 'AB' is greater than 'A'; that is, 'AB' > 'A' returns True. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If *S* is a string variable and *i* an integer expression, *S*[*i*] represents the *i*th character - or, strictly speaking, the *i*th byte in *S*. For a ShortString or AnsiString, *S*[*i*] is of type AnsiChar; for a WideString, *S*[*i*] is of type WideChar. For single-byte (Western) locales, *MyString*[2] := 'A'; assigns the value A to the second character of *MyString*. The following code uses the standard *AnsiUpperCase* function to convert *MyString* to uppercase.

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
    begin
      MyString[I] := AnsiUpperCase(MyString[I]);
      I := I - 1;
    end;
  end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as var parameters, because this results in inefficient code.

You can assign the value of a string constant - or any other expression that returns a string - to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello' + 'world';
MyString := MyString + '!';
MyString := ' '; { space }
MyString := ''; { empty string }
```

## Short Strings

---

A ShortString is 0 to 255 characters long. While the length of a ShortString can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If *S* is a ShortString variable, *Ord*(*S*[0]), like *Length*(*S*), returns the length of *S*; assigning a value to *S*[0], like calling *SetLength*, changes the length of *S*. ShortString is maintained for backward compatibility only.

The Delphi language supports short-string types - in effect, subtypes of ShortString - whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word string. For example,

```
var MyString: string[100];
```

creates a variable called `MyString` whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];  
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires - that is, the specified maximum length plus one byte. In our example, `MyString` uses 101 bytes, as compared to 256 bytes for a variable of the predefined `ShortString` type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions `High` and `Low` operate on short-string type identifiers and variables. `High` returns the maximum length of the short-string type, while `Low` returns zero.

## Long Strings

---

`AnsiString`, also called a long string, represents a dynamically allocated string whose maximum length is limited only by available memory.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty - that is, when it contains a zero-length string the pointer is nil and the string uses no additional storage. When the variable is nonempty, it points a dynamically allocated block of memory that contains the string value. The eight bytes before the location contain a 32-bit length indicator and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called reference-counting. When indexing is used to change the value of a single character in a string, a copy of the string is made if - but only if - its reference count is greater than one. This is called copy-on-write semantics.

## WideString

---

The `WideString` type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to `AnsiString`. On Win32, `WideString` is compatible with the COM `BSTR` type.

**Note:** Under Win32, `WideString` values are not reference-counted.

The Win32 platform supports single-byte and multibyte character sets as well as Unicode. With a single-byte character set (SBCS), each byte in a string represents one character.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. The null value (#0) is always a single-byte character. Multibyte character sets - especially double-byte character sets (DBCS) - are widely used for Asian languages.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called wide characters and wide character strings. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2).

The Delphi language supports single-byte and multibyte characters and strings through the Char, PChar, AnsiChar, PAnsiChar, and AnsiString types. Indexing of multibyte strings is not reliable, since `S[i]` represents the *i*th byte (not necessarily the *i*th character) in *S*. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with *Ansi*-. For example, the multibyte version of `StrPos` is `AnsiStrPos`.) Multibyte character support is operating-system dependent and based on the current locale.

Delphi supports Unicode characters and strings through the WideChar, PWideChar, and WideString types.

## Working with null-Terminated Strings

---

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on null-terminated strings. A null-terminated string is a zero-based array of characters that ends with NUL (#0); since the array has no length indicator, the first NUL character marks the end of the string. You can use Delphi constructions and special routines in the `SysUtils` unit (see Standard routines and I/O) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (`{ $X+ }`), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won't work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to #0.

## Using Pointers, Arrays, and String Constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See Pointers and pointer types.) String constants are assignment-compatible with the PChar and PWideChar types, which represent pointers to null-terminated arrays of Char and WideChar values. For example,

```
var P: PChar;
...
P := 'Hello world!'
```

points `P` to an area of memory that contains a null-terminated copy of 'Hello world!' This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!';
var P: PChar;
...
P := @TempString[0];
```

You can also pass string constants to any function that takes value or const parameters of type PChar or PWideChar - for example `StrUpper('Hello world!')`. As with assignments to a PChar, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize PChar or PWideChar constants with string literals, alone or in a structured type. Examples:

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = ('Zero', 'One', 'Two', 'Three', 'Four',
  'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with PChar and PWideChar. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

This code calls `SomeProcedure` twice with the same value.

A character pointer can be indexed as if it were an array. In the previous example, `MyPointer[0]` returns `H`. The index specifies an offset added to the pointer before it is dereferenced. (For `PWideChar` variables, the index is automatically multiplied by two.) Thus, if `P` is a character pointer, `P[0]` is equivalent to `P^` and specifies the first character in the array, `P[1]` specifies the second character in the array, and so forth; `P[-1]` specifies the 'character' immediately to the left of `P[0]`. The compiler performs no range checking on these indexes.

The `StrUpper` function illustrates the use of pointer indexing to iterate through a null-terminated string:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

## Mixing Delphi Strings and Null-Terminated Strings

You can mix long strings (`AnsiString` values) and null-terminated strings (`PChar` values) in expressions and assignments, and you can pass `PChar` values to functions or procedures that take long-string parameters. The assignment `S := P`, where `S` is a string variable and `P` is a `PChar` expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a `PChar`, the `PChar` operand is converted to a long string.

You can cast a `PChar` value as a long string. This is useful when you want to perform a string operation on two `PChar` values. For example,

```
S := string(P1) + string(P2);
```

You can also cast a long string as a null-terminated string. The following rules apply.

- If `S` is a long-string expression, `PChar(S)` casts `S` as a null-terminated string; it returns a pointer to the first character in `S`. For example, if `Str1` and `Str2` are long strings, you could call the Win32 API `MessageBox` function like this: `MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);`
- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if `S` is empty, the typecast returns `nil`.



- `PChar(S)` always returns a pointer to a memory block; if `S` is empty, a pointer to `#0` is returned.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
  - The expression cast is a long-string variable.
  - The string is not empty.
  - The string is unique - that is, has a reference count of one. To guarantee that the string is unique, call the `SetLength`, `SetString`, or `UniqueString` procedure.
  - The string has not been modified since the typecast was made.
  - The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing `WideString` values with `PWideChar` values.

# Structured Types

---

Instances of a structured type hold more than one value. Structured types include sets, arrays, records, and files as well as class, class-reference, and interface types. Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word `packed` to implement compressed data storage. For example, `type TNumbers = packed array [1..100] of Real;`

Using `packed` slows data access and, in the case of a character array, affects type compatibility (for more information, see [Memory management](#)).

This topic covers the following structured types:

- Sets
- Arrays, including static and dynamic arrays.
- Records
- File types

## Sets

---

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the base type; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

`set of baseType`

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

create a set type called `TIntSet` whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
...
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

You can also use the `set of ...` construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
...
MySet := ['a', 'b', 'c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The `in` operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by `[]`.

## Arrays

---

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

### Static Arrays

Static array types are denoted by constructions of the form

```
array[ indexType1, ..., indexTypen ] of baseType
```

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexTypes* index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexTypes*. In practice, *indexTypes* are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array [1..100] of Char;
```

declares a variable called `MyArray` that holds an array of 100 character values. Given this declaration, `MyArray[3]` denotes the third character in `MyArray`. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way `TMatrix` is declared, it represents an array of 500 real values. A variable `MyMatrix` of type `TMatrix` can be indexed like this: `MyMatrix[2,45]`; or like this: `MyMatrix[2][45]`.

Similarly,

```
packed array[Boolean, 1..10, TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions `Low` and `High` operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function `Length` returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of `Char` values is called a packed string. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See [Type compatibility and identity](#).

An array type of the form `array[0..x] of Char` is called a zero-based character array. Zero-based character arrays are used to store null-terminated strings and are compatible with `PChar` values. See [Working with null-terminated strings](#).

## Dynamic Arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the `SetLength` procedure. Dynamic-array types are denoted by constructions of the form

```
array of baseType
```

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for `MyFlexibleArray`. To create the array in memory, call `SetLength`. For example, given the previous declaration,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign nil to a variable that references the array or pass the variable to `Finalize`; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays are automatically released when their reference-count drops to zero. Dynamic arrays of length 0 have the value nil. Do not apply the dereference operator (^) to a dynamic-array variable or pass it to the `New` or `Dispose` procedure.

If `X` and `Y` are variables of the same dynamic-array type, `X := Y` points `X` to the same array as `Y`. (There is no need to allocate memory for `X` before performing this operation.) Unlike strings and static arrays, *copy-on-write* is not employed for dynamic arrays, so they are not automatically copied before they are written to. For example, after this code executes,

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

the value of `A[0]` is 2. (If `A` and `B` were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

In contrast, to make an independent copy of a dynamic array, you must use the global `Copy` function:

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := Copy(A);
  B[0] := 2; { B[0] <> A[0] }
end;
```

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
begin
```

```

    SetLength(A, 1);
    SetLength(B, 1);
    A[0] := 2;
    B[0] := 2;
end;

```

`A = B` returns False but `A[0] = B[0]` returns True.

To truncate a dynamic array, pass it to `SetLength`, or pass it to `Copy` and assign the result back to the array variable. (The `SetLength` procedure is usually faster.) For example, if `A` is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of `A`.

Once a dynamic array has been allocated, you can pass it to the standard functions `Length`, `High`, and `Low`. `Length` returns the number of elements in the array, `High` returns the array's highest index (that is, `Length - 1`), and `Low` returns 0. In the case of a zero-length array, `High` returns 1 (with the anomalous consequence that `High < Low`).

**Note:** In some function and procedure declarations, array parameters are represented as `array of baseType`, without any index types specified. For example, `function CheckStrings(A: array of string): Boolean;`

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically.

## Multidimensional Dynamic Arrays

To declare multidimensional dynamic arrays, use iterated `array of ...` constructions. For example,

```

type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;

```

declares a two-dimensional array of strings. To instantiate this array, call `SetLength` with two integer arguments. For example, if `I` and `J` are integer-valued variables,

```
SetLength(Msgs, I, J);
```

allocates an *I*-by-*J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call `SetLength`, passing it parameters for the first *n* dimensions of the array. For example,

```

var Ints: array of array of Integer;
SetLength(Ints, 10);

```

allocates ten rows for `Ints` but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of `Ints` five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column - for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the `IntToStr` function declared in the `SysUtils` unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
    begin
      SetLength(A[I], I);
      for J := Low(A[I]) to High(A[I]) do
        A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
      end;
    end;
  end;
```

## Array Types and Assignments

Arrays are assignment-compatible only if they are of the same type. Because the Delphi language uses name-equivalence for types, the following code will not compile.

```
var
  Int1: array[1..10] of Integer;
  array[1..10] of Integer;
  ...
  Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```
type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;
```

## Records

---

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
    fieldList1: type1;
    ...
    fieldListn: typen;
end
```

where *recordTypeName* is a valid identifier, each type denotes a type, and each fieldList is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called `TDateRec`.

```
type
    TDateRec = record
        Year: Integer;
        Month: (Jan, Feb, Mar, Apr, May, Jun,
                Jul, Aug, Sep, Oct, Nov, Dec);
        Day: 1..31;
    end;
```

Each `TDateRec` contains three fields: an integer value called `Year`, a value of an enumerated type called `Month`, and another integer between 1 and 31 called `Day`. The identifiers `Year`, `Month`, and `Day` are the field designators for `TDateRec`, and they behave like variables. The `TDateRec` type declaration, however, does not allocate any memory for the `Year`, `Month`, and `Day` fields; memory is allocated when you instantiate the record, like this:

```
var Record1, Record2: TDateRec;
```

This variable declaration creates two instances of `TDateRec`, called `Record1` and `Record2`.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

Or use a with statement:

```
with Record1 do
begin
    Year := 1904;
```



```

    Month := Jun;
    Day := 16;
end;

```

You can now copy the values of `Record1`'s fields to `Record2`:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the `record ...` construction directly in variable declarations:

```

var S: record
    Name: string;
    Age: Integer;
end;

```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

## Variant Parts in Records

A record type can have a variant part, which looks like a case statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```

type recordTypeName = record
    fieldList1: type1;
    ...
    fieldListn: typen;
    case tag: ordinalType of
        constantList1: (variant1);
        ...
        constantListn: (variantn);
    end;

```

The first part of the declaration - up to the reserved word `case` - is the same as that of a standard record type. The remainder of the declaration - from `case` to the optional final semicolon - is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (:) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type *ordinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantLists*.

- Each *variant* is a semicolon-delimited list of declarations resembling the *fieldList: type* constructions in the main part of the record type. That is, a variant has the form

```
fieldList1: type1;
...
fieldListn: typen;
```

where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each type denotes a type, and the final semicolon is optional. The types must not be long strings, dynamic arrays, variants (that is, Variant types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several variants which share the same space in memory. You can read or write to any field of any variant at any time; but if you write to a field in one variant and then to a field in another variant, you may be overwriting your own data. The tag, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type
  TEmployee = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Salaried: Boolean of
      True: (AnnualSalary: Currency);
      False: (HourlyWage: Currency);
    end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of `TEmployee`, there is no reason to allocate enough memory for both fields. In this case, the only difference between the variants is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```
type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
      True: (Birthplace: string[40]);
      False: (Country: string[20];
              EntryPort: string[20];
              EntryDate, ExitDate: TDate);
```

```

end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
    end;
end;

```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest variant. The optional tag and the *constantLists* (like [Rectangle](#), [Triangle](#), and so forth in the last example) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit Real as the first field in one variant and a 32-bit Integer as the first field in another, you can assign a value to the Real field and then read back the first 32 bits of it as the value of the Integer field (passing it, say, to a function that requires integer parameters).

## File Types

---

A file is a sequence of elements of the same type. Standard I/O routines use the predefined [TextFile](#) or [Text](#) type, which represents a file containing characters organized into lines. For more information about file input and output, see Standard routines and I/O.

To declare a file type, use the syntax

```
type fileTypeName = filetype
```

where *fileTypeName* is any valid identifier and type is a fixed-size type. Pointer types - whether implicit or explicit - are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```

type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;

```

```
end;  
PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the `file of ...` construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word `file` by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see [Untyped files](#).

Files are not allowed in arrays or records.

# Pointers and Pointer Types

---

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it points to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure. If that address is already taken, then the pointer holds the address to the first element.

Pointers are typed to indicate the kind of data stored at the addresses they hold. The general-purpose Pointer type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

This topic contains information on the following:

- General overview of pointer types.
- Declaring and using the pointer types supported by Delphi.

## Overview of pointers

---

To see how pointers work, look at the following example.

```
1      var
2          X, Y: Integer; // X and Y are Integer variables
3          P: ^Integer    // P points to an Integer
4      begin
5          X := 17;        // assign a value to X
6          P := @X;        // assign the address of X to P
7          Y := P^;        // dereference P; assign the result to Y
8      end;
```

Line 2 declares `X` and `Y` as variables of type `Integer`. Line 3 declares `P` as a pointer to an `Integer` value; this means that `P` can point to the location of `X` or `Y`. Line 5 assigns a value to `X`, and line 6 assigns the address of `X` (denoted by `@X`) to `P`. Finally, line 7 retrieves the value at the location pointed to by `P` (denoted by `P^`) and assigns it to `Y`. After this code executes, `X` and `Y` have the same value, namely 17.

The `@` operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see The `@` operator and Procedural types in statements and expressions.

The symbol `^` has two purposes, both of which are illustrated in our example. When it appears before a type identifier

`^typeName`

it denotes a type that represents pointers to variables of type `typeName`. When it appears after a pointer variable

*pointer*<sup>^</sup>

it dereferences the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another - something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand the Delphi language, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class instance variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Delphi's strict data typing. By referencing a variable with an all-purpose `Pointer`, casting the `Pointer` to a more specific type, and then dereferencing it, you can treat the data stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from `R` to `I`, without converting it.

In addition to assigning the result of an `@` operation, you can use several standard routines to give a value to a pointer. The `New` and `GetMem` procedures assign a memory address to an existing pointer, while the `Addr` and `Ptr` functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `PI^.Data^`.

The reserved word `nil` is a special constant that can be assigned to any pointer. When `nil` is assigned to a pointer, the pointer doesn't reference anything.

## Pointer Types

---

You can declare a pointer to any type, using the syntax

```
type pointerTypeName = ^type
```

When you define a record or other data type, it's a common practice also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Standard pointer types exist for many purposes. The most versatile is `Pointer`, which can point to data of any kind. But a `Pointer` variable cannot be dereferenced; placing the `^` symbol after a `Pointer` variable causes a compilation error. To access the data referenced by a `Pointer` variable, first cast it to another pointer type and then dereference it.

## Character Pointers

The fundamental types `PAnsiChar` and `PWideChar` represent pointers to `AnsiChar` and `WideChar` values, respectively. The generic `PChar` represents a pointer to a `Char` (that is, in its current implementation, to an `AnsiChar`). These character pointers are used to manipulate null-terminated strings. (See [Working with null-terminated strings](#).)

## Type-checked Pointers

The `$T` compiler directive controls the types of pointer values generated by the `@` operator. This directive takes the form of:

```
{ $T+ } or { $T- }
```

In the `{ $T- }` state, the result type of the `@` operator is always an untyped pointer that is compatible with all other pointer types. When `@` is applied to a variable reference in the `{ $T+ }` state, the type of the result is `^T`, where `T` is compatible only with pointers to the type of the variable.

## Other Standard Pointer Types

The `System` and `SysUtils` units declare many standard pointer types that are commonly used.

### *Selected pointer types declared in `System` and `SysUtils`*

Pointer type	Points to variables of type
<code>PAnsiString</code> , <code>PString</code>	<code>AnsiString</code>
<code>PByteArray</code>	<code>TByteArray</code> (declared in <code>SysUtils</code> ). Used to typecast dynamically allocated memory for array access.
<code>PCurrency</code> , <code>PDouble</code> , <code>PExtended</code> , <code>PSingleCurrency</code> , <code>Double</code> , <code>Extended</code> , <code>Single</code>	
<code>PInteger</code>	<code>Integer</code>
<code>POleVariant</code>	<code>OleVariant</code>
<code>PShortString</code>	<code>ShortString</code> . Useful when porting legacy code that uses the old <code>PString</code> type.
<code>PTextBuf</code>	<code>TTextBuf</code> (declared in <code>SysUtils</code> ). <code>TTextBuf</code> is the internal buffer type in a <code>TTextRec</code> file record.)

PVarRec	TVarRec (declared in <a href="#">System</a> )
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray (declared in <a href="#">SysUtils</a> ). Used to typecast dynamically allocated memory for arrays of 2-byte values.



# Procedural Types

---

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.

## About Procedural Types

---

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called `Calc` that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the `Calc` function to the variable `F`:

```
var F: function(X,Y: Integer): Integer;  
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word procedure or function, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the previous example) or to declare new types:

```
type  
  TIntegerFunction = function: Integer;  
  TProcedure = procedure;  
  TStrProc = procedure(const S: string);  
  TMathFunc = function(X: Double): Double;  
var  
  F: TIntegerFunction;    { F is a parameterless function that returns an  
integer }  
  Proc: TProcedure;      { Proc is a parameterless procedure }  
  SP: TStrProc;           { SP is a procedure that takes a string parameter }  
  M: TMathFunc;           { M is a function that takes a Double (real)  
parameter and returns a Double }  
  
  procedure FuncProc(P: TIntegerFunction); { FuncProc is a procedure whose  
only parameter is a parameterless integer-valued function }
```

On Win32, the variables shown in the previous example are all procedure pointers - that is, pointers to the address of a procedure or function. On the .NET platform, procedural types are implemented as delegates. If you want to reference a method of an instance object (see Classes and objects), you need to add the words `of object` to the procedural type name. For example

```
type
```

```
TMethod      = procedure of object;
TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent method pointers. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ...
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
  - the same return value (or no return value), and
  - the same number of parameters, with identically typed parameters in corresponding positions.
- (Parameter names do not matter.)

Procedure pointer types are always incompatible with method pointer types. The value nil can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like [Length](#) as a procedural value, write a wrapper for it:

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

## Procedural Types in Statements and Expressions

---

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```

var
  F: function(X: Integer): Integer;
  I: Integer;
  function SomeFunction(X: Integer): Integer;
  ...
  F := SomeFunction;    // assign SomeFunction to F
  I := F(4);           // call function; assign result to I

```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```

var
  F, G: function: Integer;
  I: Integer;
  function SomeFunction: Integer;
  ...
  F := SomeFunction;    // assign SomeFunction to F
  G := F;               // copy F to G
  I := G;               // call function; assign result to I

```

The first statement assigns a procedural value to `F`. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to `I`. Because `I` is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of `F` results in a function call; the compiler calls the function pointed to by `F`, then calls the function `MyFunction`, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where `F` references a procedure (which doesn't return a value), or where `F` references a function that requires parameters, the previous statement causes a compilation error. To compare the procedural value of `F` with `MyFunction`, use

```
if @F = @MyFunction then ...;
```

`@F` converts `F` into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of `MyFunction`.

To get the memory address of a procedural variable (rather than the address stored in it), use `@@`. For example, `@@F` returns the address of `F`.

The `@` operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```
var StrComp: function(Str1, Str2: PChar): Integer;  
...  
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

calls the [GetProcAddress](#) function and points [StrComp](#) to the result.

Any procedural variable can hold the value nil, which means that it points to nothing. But attempting to call a nil-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function [Assigned](#):

```
if Assigned(OnClick) then OnClick(X);
```

# Variant Types

---

This topic discusses the use of variant data types.

## Variants Overview

---

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type `Variant`, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See [Object interfaces](#).) They can hold dynamic arrays, and they can hold a special kind of static array called a variant array. (See [Variant arrays](#).) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if `v` is a variant that holds a string value, the construction `v[1]` causes a runtime error.

You can define custom Variants that extend the `Variant` type to hold arbitrary values. For example, you can define a `Variant` string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the `TCustomVariantType` class.

**Note:** This, and almost all variant functionality, is implemented in the [Variants](#) unit.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value `Unassigned`. The special value `Null` indicates unknown or missing data.

The standard function `VarType` returns a variant's type code. The `varTypeMask` constant is a bit mask used to extract the code from `VarType`'s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns `True` if `v` contains a `Double` or an array of `Double`. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The `TVarData` record type defined in the [System](#) unit can be used to typecast variants and gain access to their internal representation.

## Variant Type Conversions

---

All integer, real, string, character, and Boolean types are assignment-compatible with `Variant`.

Expressions can be explicitly cast as variants, and the `VarAsType` and `VarCast` standard routines can

be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1;           { integer value }
  V2 := 1234.5678; { real value }
  V3 := 'Hello world!'; { string value }
  V4 := '1000';      { string value }
  V5 := V1 + V2 + V4; { real value 2235.5678}
  I := V1;           { I = 1 (integer value) }
  D := V2;           { D = 1234.5678 (real value) }
  S := V3;           { S = 'Hello world!' (string value) }
  I := V4;           { I = 1000 (integer value) }
  S := V5;           { S = '2235.5678' (string value) }
end;
```

The compiler performs type conversions according to the following rules.

#### **Variant type conversion rules**

Target	integer	real	string	Boolean
<b>Source</b>				
integer	converts integer formats	converts to real	converts to string representation	returns False if 0, True otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	returns False if 0, True otherwise
string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/character formats	returns False if string is 'false' (noncase-sensitive) or a numeric string that evaluates to 0, True if string is 'true' or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string (above)
Boolean	False = 0, True = 1 (255 if Byte)	False = 0, True = 1	False = '0', True = '1'	False = False, True = True
<b>Unassigned</b>	returns 0	returns 0	returns empty string	returns False
<b>Null</b>	raises exception	raises exception	raises exception	raises exception

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid variant operations, assignments or casts raise an `EVariantError` exception or an exception class descending from `EVariantError`.

Special conversion rules apply to the `TDateTime` type declared in the `System` unit. When a `TDateTime` is converted to any other type, it is treated as a normal `Double`. When an integer, real, or Boolean is converted to a `TDateTime`, it is first converted to a `Double`, then read as a date-time value. When a string is converted to a `TDateTime`, it is interpreted as a date-time value using the regional settings. When an `Unassigned` value is converted to `TDateTime`, it is treated like the real or integer value 0. Converting a `Null` value to `TDateTime` raises an exception.

On the Win32 platform, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

## Variants in Expressions

---

All operators except `^`, `is`, and `in` take variant operands. Except for comparisons, which always return a Boolean result, any operation on a variant value returns a variant result. If an expression combines variants with statically-typed values, the statically-typed values are automatically converted to variants.

This is not true for comparisons, where any operation on a `Null` variant produces a `Null` variant. For example:

```
V := Null + 3;
```

assigns a `Null` variant to `V`. By default, comparisons treat the `Null` variant as a unique value that is less than any other value. For example:

```
if Null > -3 then ... else ...;
```

In this example, the `else` part of the `if` statement will be executed. This behavior can be changed by setting the `NullEqualityRule` and `NullMagnitudeRule` global variables.

## Variant Arrays

---

You cannot assign an ordinary static array to a variant. Instead, create a variant array by calling either of the standard functions `VarArrayCreate` or `VarArrayOf`. For example,

```
V: Variant;  
...  
V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant `V`. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a `var` parameter. Variant arrays are always indexed with integers.

The second parameter in the call to `VarArrayCreate` is the type code for the array's base type. For a list of these codes, see `VarType`. Never pass the code `varString` to `VarArrayCreate`; to create a variant array of strings, use `varOleStr`.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except `ShortString` and `AnsiString`, and if the base type of the array is `Variant`, its elements can even be heterogeneous. Use the `VarArrayRedim` function to resize a variant array. Other standard routines that operate on variant arrays include `VarArrayDimCount`, `VarArrayLowBound`, `VarArrayHighBound`, `VarArrayRef`, `VarArrayLock`, and `VarArrayUnlock`.

**Note:** Variant arrays of custom variants are not supported, as instances of custom variants can be added to a `VarVariant` variant array.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

## OleVariant

---

The `OleVariant` type exists on both the Windows and Linux platforms. The main difference between *Variant* and `OleVariant` is that *Variant* can contain data types that only the current application knows what to do with. `OleVariant` can only contain the data types defined as compatible with OLE Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Delphi string, or a one of the new custom variant types) to an `OleVariant`, the runtime library tries to convert the *Variant* into one of the `OleVariant` standard data types (such as a Delphi string converts to an OLE BSTR string). For example, if a variant containing an `AnsiString` is assigned to an `OleVariant`, the `AnsiString` becomes a `WideString`. The same is true when passing a *Variant* to an `OleVariant` function parameter.



# Type Compatibility and Identity

---

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include:

- Type identity
- Type compatibility
- Assignment compatibility

## Type Identity

---

When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

`T1`, `T2`, `T3`, `T4`, and `Integer` all denote the same type. To create distinct types, repeat the word `type` in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called `TMyInteger` which is not identical to `Integer`.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, `TS1` and `TS2`. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```

type MyString = string[10];
var
    S1: MyString;
    S2: MyString;

```

## Type Compatibility

---

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of characters.
- One is a string type and the other is a string, packed-string, or Char type.
- One type is Variant and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is PChar or PWideChar and the other is a zero-based character array of the form `array [0..n] of PChar or PWideChar`.
- One type is Pointer (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the `{ $T+ }` compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

## Assignment Compatibility

---

Assignment-compatibility is not a symmetric relation. An expression of type T2 can be assigned to a variable of type T1 if the value of the expression falls in the range of T1 and at least one of the following conditions is satisfied.

- T1 and T2 are of the same type, and it is not a file type or structured type that contains a file type at any level.
- T1 and T2 are compatible ordinal types.
- T1 and T2 are both real types.
- T1 is a real type and T2 is an integer type.
- T1 is PChar, PWideChar or any string type and the expression is a string constant.
- T1 and T2 are both string types.
- T1 is a string type and T2 is a Char or packed-string type.
- T1 is a long string and T2 is PChar or PWideChar.

- T1 and T2 are compatible packed-string types.
- T1 and T2 are compatible set types.
- T1 and T2 are compatible pointer types.
- T1 and T2 are both class, class-reference, or interface types and T2 is a derived from T1.
- T1 is an interface type and T2 is a class type that implements T1.
- T1 is PChar or PWideChar and T2 is a zero-based character array of the form `array[0..n] of Char` (when T1 is PChar) or of `WideChar` (when T1 is PWideChar).
- T1 and T2 are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type.)
- T1 is Variant and T2 is an integer, real, string, character, Boolean, interface type or OleVariant type.
- T1 is an OleVariant and T2 is an integer, real, string, character, Boolean, interface, or Variant type.
- T1 is an integer, real, string, character, or Boolean type and T2 is Variant or OleVariant.
- T1 is the `IUnknown` or `IDispatch` interface type and T2 is Variant or OleVariant. (The variant's type code must be `varEmpty`, `varUnknown`, or `varDispatch` if T1 is `IUnknown`, and `varEmpty` or `varDispatch` if T1 is `IDispatch`.)

# Declaring Types

---

This topic describes the syntax of Delphi type declarations.

## Type Declaration Syntax

---

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

```
type newTypeName = type
```

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier's scope doesn't include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

*X* and *Y* are of the same type; at runtime, there is no way to distinguish *TValue* from *Real*. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information for example, to associate a property editor with properties of a particular type - the distinction between 'different name' and 'different type' becomes important. In this case, use the syntax

```
type newTypeName = type type
```

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called *TValue*.

For *var* parameters, types of formal and actual must be identical. For example,

```
type
  TMyType = type Integer
  procedure p(var t:TMyType);
  begin
  end;
```

```
procedure x;  
var  
  m: TMyType;  
  i: Integer;  
begin  
  p(m); // Works  
  p(i); // Error! Types of formal and actual must be identical.  
end;
```

**Note:** This only applies to `var` parameters, not to `const` or by-value parameters.

# Variables

---

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

## Declaring Variables

---

The basic syntax for a variable declaration is

```
var identifierList : type;
```

where *identifierList* is a comma-delimited list of valid identifiers and *type* is any valid type. For example,

```
var I: Integer;
```

declares a variable `I` of type Integer, while

```
var X, Y: Real;
```

declares two variables - `X` and `Y` - of type Real.

Consecutive variable declarations do not have to repeat the reserved word `var`:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called local, while other variables are called global. Global variables can be initialized at the same time they are declared, using the syntax

```
var identifier: type = constantExpression;
```

where *constantExpression* is any constant expression representing a value of type *type*. Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
...
I := 7;
```

Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and their contents are undefined until a value is assigned to them.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see [Memory management](#).

## Absolute Addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive `absolute` after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

specifies that the variable `StrLen` should start at the same address as `Str`. Since the first byte of a short string contains the string's length, the value of `StrLen` is the length of `Str`.

You cannot initialize a variable in an absolute declaration or combine `absolute` with any other directives.

## Dynamic Variables

You can create dynamic variables by calling the `GetMem` or `New` procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use `FreeMem` to destroy variables created by `GetMem` and `Dispose` to destroy variables created by `New`. Other standard routines that operate on dynamic variables include `ReallocMem`, `AllocMem`, `Initialize`, `Finalize`, `StrAlloc`, and `StrDispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

## Thread-local Variables

Thread-local (or thread) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with `threadvar` instead of `var`. For example,

```
threadvar X: Integer;
```

#### Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the absolute directive.

Dynamic variables that are ordinarily managed by the compiler (long strings, wide strings, dynamic arrays, variants, and interfaces) can be declared with `threadvar`, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory from within the thread, before the thread terminates. For example,

```
threadvar S: AnsiString;  
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';  
...  
S := ''; // free the memory used by S
```

**Note:** Use of such constructs is discouraged.

You can free a variant by setting it to `Unassigned` and an interface or dynamic array by setting it to `nil`.



## Declared Constants

Several different language constructions are referred to as 'constants'. There are numeric constants (also called numerals) like 17, and string constants (also called character strings or string literals) like 'Hello world!'. Every enumerated type defines constants that represent the values of that type. There are predefined constants like True, False, and nil. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

## True Constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called `MaxValue` that returns the integer 237. The syntax for declaring a true constant is

```
const identifier = constantExpression
```

where *identifier* is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program.

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called `MyNumber`, of type `Int64`, that returns the integer 17. Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If *constantExpression* is a real, its type is Extended. If it is an integer, its type is given by the table below.

### Types for integer constants

Range of constant(hexadecimal)	Range of constant(decimal)	Type
-\$8000000000000000..-\$80000001	-2^63..-2147483649	Int64
-\$80000000..-\$8001	-2147483648..-32769	Integer
-\$8000..-\$81	-32768..-129	Smallint
-\$80..-1	-128..-1	Shortint

0..\$7F	0..127	0..127
\$80..\$FF	128..255	Byte
\$0100..\$7FFF	256..32767	0..32767
\$8000..\$FFFF	32768..65535	Word
\$10000..\$7FFFFFFF	65536..2147483647	0..2147483647
\$80000000..\$FFFFFFFF	2147483648..4294967295	Cardinal
\$100000000..\$FFFFFFFFFFFFFFFF	4294967296..2 <sup>63</sup> -1	Int64

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

## Constant Expressions

A constant expression is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants True, False, and nil; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

```
AbsHigh  LowPred  Succ
ChrLengthOddRound Swap
Hi Lo      OrdSizeOfTrunc
```

This definition of a constant expression is used in several places in Delphi's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing case statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

## Resource Strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program.

Resource strings are declared like other true constants, except that the word `const` is replaced by `resourcestring`. The expression to the right of the `=` symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks';
  SomeResourceString = SomeTrueConstant;
```

## Typed Constants

---

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

Declare a typed constant like this:

*const identifier: type = value*

where *identifier* is any valid identifier, *type* is any type except files and variants, and *value* is an expression of type *type*. For example,

```
const Max: Integer = 100;
```

In most cases, *value* must be a constant expression; but if *type* is an array, record, procedural, or pointer type, special rules apply.

## Array Constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called `Digits` that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the previous declaration can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;  
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

creates an array called `Maze` where

```
Maze[0,0,0] = 0  
Maze[0,0,1] = 1  
Maze[0,1,0] = 2  
Maze[0,1,1] = 3  
Maze[1,0,0] = 4  
Maze[1,0,1] = 5  
Maze[1,1,0] = 6  
Maze[1,1,1] = 7
```

Array constants cannot contain file-type values at any level.

## Record Constants

To declare a record constant, specify the value of each field - as `fieldName: value`, with the field assignments separated by semicolons - in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
```

```

TPoint = record
    X, Y: Single;
end;
TVector = array[0..1] of TPoint;
TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
end;
const
    Origin: TPoint = (X: 0.0; Y: 0.0);
    Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
    SomeDay: TDate = (D: 2; M: Dec; Y: 1960);

```

Record constants cannot contain file-type values at any level.

## Procedural Constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```

function Calc(X, Y: Integer): Integer;
begin
    ...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;

```

Given these declarations, you can use the procedural constant `MyFunction` in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value `nil` to a procedural constant.

## Pointer Constants

When you declare a pointer constant, you must initialize it to a value that can be resolved at least as a relative address at compile time. There are three ways to do this: with the `@` operator, with `nil`, and (if the constant is of type `PChar` or `PWideChar`) with a string literal. For example, if `I` is a global variable of type `Integer`, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a PChar constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

# Procedures and Functions

---

This section describes the syntax of function and procedure declarations.

# Procedures and Functions

---

This topic covers the following items:

- Declaring procedures and functions
- Calling conventions
- Forward and interface declarations
- Declaration of external routines
- Overloading procedures and functions
- Local declarations and nested routines

## About Procedures and Functions

---

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls `SomeFunction` and assigns the result to `I`. Function calls cannot appear on the left side of an assignment statement.

Procedure calls - and, when extended syntax is enabled (`{ $X+ }`), function calls - can be used as complete statements. For example,

```
DoSomething;
```

calls the `DoSomething` routine; if `DoSomething` is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

## Declaring Procedures and Functions

---

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the prototype, heading, or header. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's body or block.

## Procedure Declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
```



```

    localDeclarations;
begin
    statements
end;

```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and (*parameterList*), *directives*, and *localDeclarations*; are optional.

Here is an example of a procedure declaration:

```

procedure NumString(N: Integer; var S: string);
var
    V: Integer;
begin
    V := Abs(N);
    S := '';
    repeat
        S := Chr(V mod 10 + Ord('0')) + S;
        V := V div 10;
    until V = 0;
    if N < 0 then S := '-' + S;
end;

```

Given this declaration, you can call the `NumString` procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value '17' to `MyString` (which must be a string variable).

Within a procedure's statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like `N` and `S` in the previous example); the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

## Function Declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```

function functionName(parameterList): returnType; directives;
    localDeclarations;
begin
    statements
end;

```

where *functionName* is any valid identifier, *returnType* is a type identifier, *statements* is a sequence of statements that execute when the function is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable `Result`.

As long as extended syntax is enabled (`{ $X+ }`), `Result` is implicitly declared in every function. Do not try to redeclare it.

For example,

```
function WF: Integer;
begin
    WF := 17;
end;
```

defines a constant function called `WF` that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
    Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
    X: Real;
    I: Integer;
begin
    X := A[0];
    for I := 1 to N - 1 do
        if X < A[I] then X := A[I];
    Max := X;
end;
```

You can assign a value to `Result` or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to `Result` or to the function name becomes the function's return value. For example,

```

function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;

```

**Result** and the function name always represent the same value. Hence

```

function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;

```

returns the value 11. But **Result** is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like **Result**) to track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. **Result**, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

If the function exits without assigning a value to **Result** or the function name, then the function's return value is undefined.

## Calling Conventions

---

When you declare a procedure or function, you can specify a calling convention using one of the directives `register`, `pascal`, `cdecl`, `stdcall`, and `safecall`. For example,

```

function MyFunction(X, Y: Real): Real; cdecl;

```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is `register`.

- The register and pascal conventions pass parameters from left to right; that is, the left most parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The cdecl, stdcall, and safecall conventions pass parameters from right to left.
- For all conventions except cdecl, the procedure or function removes parameters from the stack upon returning. With the cdecl convention, the caller removes parameters from the stack when the call returns.
- The register convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The safecall convention implements exception 'firewalls.' On Win32, this implements interprocess COM error notification.

The table below summarizes calling conventions.

### ***Calling conventions***

Directive	Parameter order	Clean-up	Passes parameters in registers?
<b>register</b>	Left-to-right	Routine	Yes
<b>pascal</b>	Left-to-right	Routine	No
<b>cdecl</b>	Right-to-left	Caller	No
<b>stdcall</b>	Right-to-left	Routine	No
<b>safecall</b>	Right-to-left	Routine	No

The default register convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use register.) The cdecl convention is useful when you call functions from shared libraries written in C or C++, while stdcall and safecall are recommended, in general, for calls to external code. On Win32, the operating system APIs are stdcall and safecall. Other operating systems generally use cdecl. (Note that stdcall is more efficient than cdecl.)

The safecall convention must be used for declaring dual-interface methods. The pascal convention is maintained for backward compatibility.

The directives near, far, and export refer to calling conventions in 16-bit Windows programming. They have no effect in Win32, or in .NET applications and are maintained for backward compatibility only.

## **Forward and Interface Declarations**

The forward directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called `Calculate`. Somewhere after the forward declaration, the routine must be redeclared in a defining declaration that includes a block. The defining declaration for `Calculate` might look like this:

```
function Calculate;
... { declarations }
begin
... { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the forward declaration exactly (except that default parameters can be omitted). If the forward declaration specifies an overloaded procedure or function, then the defining declaration must repeat the parameter list.

A forward declaration and its defining declaration must appear in the same type declaration section. That is, you can't add a new section (such as a var section or const section) between the forward declaration and the defining declaration. The defining declaration can be an external or assembler declaration, but it cannot be another forward declaration.

The purpose of a forward declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the forward-declared routine before it is actually defined. Besides letting you organize your code more flexibly, forward declarations are sometimes necessary for mutual recursions.

The forward directive has no effect in the interface section of a unit. Procedure and function headers in the interface section behave like forward declarations and must have defining declarations in the implementation section. A routine declared in the interface section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

## External Declarations

---

The external directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C function that takes a variable number of parameters, use the varargs directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The varargs directive works only with external routines and only with the cdecl calling convention.

## Linking to Object Files

To call routines from a separately compiled object file, first link the object file to your application using the `$L` (or `$LINK`) compiler directive. For example,

```
{ $L BLOCK.OBJ }
```

links BLOCK.OBJ into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;  
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the `MoveWord` and `FillWord` routines from BLOCK.OBJ.

On the Win32 platform, declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Delphi source code.

## Importing Functions from Libraries

To import routines from a dynamically loadable library (.DLL), attach a directive of the form

```
external stringConstant;
```

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Win32

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called `SomeFunction` from `strlib.dll`.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the external directive:

```
external stringConstant1 name stringConstant2;
```

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine's original name.

The following declaration imports a function from `user32.dll` (part of the Win32 API).

```
function MessageBox(HWND: Integer; Text, Caption: PChar; Flags: Integer):  
Integer; stdcall; external 'user32.dll' name 'MessageBoxA';
```

The function's original name is `MessageBoxA`, but it is imported as `MessageBox`.

Instead of a name, you can use a number to identify the routine you want to import:

```
external stringConstant index integerConstant;
```

where *integerConstant* is the routine's index in the export table.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

## Overloading Procedures and Functions

---

You can declare more than one routine in the same scope with the same name. This is called overloading. Overloaded routines must be declared with the `overload` directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
    Result := X/Y;
end

function Divide(X, Y: Integer): Integer; overload;
begin
    Result := X div Y;
end;
```

These declarations create two functions, both called `Divide`, that take parameters of different types. When you call `Divide`, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first `Divide` function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types - for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type `Extended`.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
...
procedure Cap(var Str: string); overload;
...
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
...
```

```
function Func(X: Integer; Y: Real): Real; overload;
...

```

are legal.

When an overloaded routine is declared in a forward or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain AnsiString/PChar and WideString/WideChar parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is AnsiString/PChar.

```
procedure test(const S: String); overload;
procedure test(const W: WideString); overload;
var
  a: string;
  b: widestring;
begin
  a := 'a';
  b := 'b';
  test(a);      // calls String version
  test(b);      // calls WideString version
  test('abc');  // calls String version
  test(WideString('abc')); // calls widestring version
end;

```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);      // integer version
  foo(v);      // variant version

```



```
foo(1.2);      // variant version (float literals -> extended precision)
end;
```

This example calls the variant version of `foo`, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. Extended is also not an exact match for Variant, but Variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const d: double = 1.2;
begin
    foo(d);
end;
```

The above code works correctly, and calls the double version.

```
const s: single = 1.2;
begin
    foo(s);
end;
```

The above code also calls the double version of `foo`. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful not to introduce ambiguous parameter signatures.

You can limit the potential effects of overloading by qualifying a routine's name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in `Unit1`; if no routine in `Unit1` matches the name and parameter list in the call, an error results.

## Local Declarations

---

The body of a function or procedure often begins with declarations of local variables used in the routine's statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

## Nested Routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called `DoSomething` contains a nested procedure.

```
procedure DoSomething(S: string);
var
    X, Y: Integer;

procedure NestedProc(S: string);
begin
    ...
end;

begin
    ...
    NestedProc(S);
    ...
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, `NestedProc` can be called only within `DoSomething`.

For real examples of nested routines, look at the `DateTimeToString` procedure, the `ScanDate` function, and other routines in the `SysUtils` unit.

# Parameters

---

This topic covers the following items:

- Parameter semantics
- String parameters
- Array parameters
- Default parameters

## About Parameters

---

Most procedure and function headers include a parameter list. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the = symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by `var`, `const`, or `out`. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWND: Integer; Text, Caption: PChar; Flags: Integer)
(const P: I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
    ...
end;
```

Within the procedure or function body, the parameter names (`x` and `y` in the first example) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

## Parameter Semantics

---

Parameters are categorized in several ways:

- Every parameter is classified as value, variable, constant, or out. Value parameters are the default; the reserved words `var`, `const`, and `out` indicate variable, constant, and out parameters, respectively.
- Value parameters are always typed, while constant, variable, and out parameters can be either typed or untyped.
- Special rules apply to array parameters.

Files and instances of structured types that contain files can be passed only as variable (`var`) parameters.

## Value and Variable Parameters

Most parameters are either value parameters (the default) or variable (`var`) parameters. Value parameters are passed by value, while variable parameters are passed by reference. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;    // X is a value parameter
begin
    X := X * 2;
    Result := X;
end;
```

```
function DoubleByRef(var X: Integer): Integer;  // X is a variable parameter
begin
    X := X * 2;
    Result := X;
end;
```

These functions return the same result, but only the second one - `DoubleByRef` can change the value of a variable passed to it. Suppose we call the functions like this:

```
var
    I, J, V, W: Integer;
begin
    I := 4;
    V := 4;
    J := DoubleByValue(I);    // J = 8, I = 4
    W := DoubleByRef(V);      // W = 8, V = 8
end;
```

After this code executes, the variable `I`, which was passed to `DoubleByValue`, has the same value we initially assigned to it. But the variable `V`, which was passed to `DoubleByRef`, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more var parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
    X := X + 1;
    Y := Y + 1;
end;

var I: Integer;
begin
    I := 1;
    AddOne(I, I);
end;
```

After this code executes, the value of `I` is 3.

If a routine's declaration specifies a var parameter, you must pass an assignable expression - that is, a variable, typed constant (in the `{SJ+}` state), dereferenced pointer, field, or indexed variable to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in var parameters - for example, `DoubleByRef(MyArray[I])` - are evaluated once, before execution of the routine.

## Constant Parameters

A constant (`const`) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a var parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using `const` allows the compiler to optimize code for structured - and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the `CompareStr` function in the `SysUtils` unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because `S1` and `S2` are not modified in the body of `CompareStr`, they can be declared as constant parameters.

## Out Parameters

An out parameter, like a variable parameter, is passed by reference. With an out parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The out parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call `GetInfo`, you must pass it a variable of type `SomeRecordType`:

```
var MyRecord: SomeRecordType;  
...  
GetInfo(MyRecord);
```

But you're not using `MyRecord` to pass any data to the `GetInfo` procedure; `MyRecord` is just a container where you want `GetInfo` to store the information it generates. The call to `GetInfo` immediately frees the memory used by `MyRecord`, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use out parameters when you pass an uninitialized variable to a function or procedure.

## Untyped Parameters

You can omit type specifications when declaring `var`, `const`, and out parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called `TakeAnything` that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called `Equal` that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;  
type  
  TBytes = array[0..MaxInt - 1] of Byte;  
var  
  N : Integer;  
begin  
  N := 0;
```

```

while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
  Inc(N);
  Equal := N = Size;
end;

```

Given the declarations

```

type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;

```

you could make the following calls to `Equal`:

```

Equal(Vec1, Vec2, SizeOf(TVector));      // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N);  // compare first N elements of Vec1
and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5); // compare first 5 to last
5 elements of Vec1
Equal(Vec1[1], P, 4);                     // compare Vec1[1] to P.X and Vec1
[2] to P.Y

```

## String Parameters

---

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```

procedure Check(S: string[20]); // syntax error

```

causes a compilation error. But

```

type TString20 = string[20];
procedure Check(S: TString20);

```

is valid. The special identifier `OpenString` can be used to declare routines that take short-string parameters of varying length:

```

procedure Check(S: OpenString);

```

When the `{ $H }` and `{ $P+ }` compiler directives are both in effect, the reserved word `string` is equivalent to `OpenString` in parameter declarations.

Short strings, `OpenString`, `$H`, and `$P` are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

## Array Parameters

---

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer) // syntax error<
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. Another approach is to use open array parameters.

Since the Delphi language does not implement value semantics for dynamic arrays, 'value' parameters in routines do not represent a full copy of the dynamic array. In this example

```
type
  TDynamicArray = array of Integer;
procedure p(Value: TDynamicArray);
begin
  Value[0] := 1;
end;

procedure Run;
var
  a: TDynamicArray;
begin
  SetLength(a, 1);
  a[0] := 0;
  p(a);
  Writeln(a[0]); // Prints '1'
end;
```

Note that the assignment to `Value[0]` in routine `p` will modify the content of dynamic array of the caller, despite `Value` being a by-value parameter. If a full copy of the dynamic array is required, use the `Copy` standard procedure to create a value copy of the dynamic array.

## Open Array Parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of type` (rather than `array [X..Y] of type`) in the parameter declaration. For example,



```
function Find(A: array of Char): Integer;
```

declares a function called `Find` that takes a character array of any size and returns an integer.

**Note:** The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The previous example creates a function that takes any array of Char elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;  
function Find(A: TDynamicCharArray): Integer;
```

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard `Low` and `High` functions return 0 and `Length-1`, respectively. The `SizeOf` function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped var parameters. They cannot be passed to `SetLength`.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a `Clear` procedure that assigns zero to each element in an array of reals and a `Sum` function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);  
var  
    I: Integer;  
begin  
    for I := 0 to High(A) do A[I] := 0;  
end;  
  
function Sum(const A: array of Real): Real;  
var  
    I: Integer;  
    S: Real;  
begin  
    S := 0;  
    for I := 0 to High(A) do S := S + A[I];  
    Sum := S;  
end;
```

When you call routines that use open array parameters, you can pass open array constructors to them.

## Variant Open Array Parameters

Variant open array parameters allow you to pass an array of differently typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify `array of const` as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called `DoSomething` that can operate on heterogeneous arrays.

The `array of const` construction is equivalent to `array of TVarRec`. `TVarRec`, declared in the `System` unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. `TVarRec`'s `VType` field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as `Pointer` and must be typecast to `string`.

The following example uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in `SysUtils`.

```
function MakeStr(const Args: array of const): string
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:  Result := Result + IntToStr(VInteger);
        vtBoolean:  Result := Result + BoolToStr(VBoolean);
        vtChar:     Result := Result + VChar;
        vtExtended: Result := Result + FloatToStr(VExtended^);
        vtString:   Result := Result + VString^;
        vtPChar:    Result := Result + VPChar;
        vtObject:   Result := Result + VObject.ClassName;
        vtClass:    Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:  Result := Result + CurrToStr(VCurrency^);
        vtVariant:   Result := Result + string(VVariant^);
        vtInt64:     Result := Result + IntToStr(VInt64^);
      end;
    end;
  end;
```

We can call this function using an open array constructor. For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string `'test100 T3.14159TForm'`.

## Default Parameters

---

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed const and value parameters. To provide a default value, end the parameter declaration with the `=` symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);  
    FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real; // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string); // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;  
function Resizer(X: Real; Y: Real = 2.0): Real;  
var  
    F: TResizer;  
    N: Real;
```

the statements

```
F := Resizer;  
F(N);
```

result in the values `(N, 1.0)` being passed to `Resizer`.

Default parameters are limited to values that can be specified by a constant expression. Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than nil as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

## Default Parameters and Overloaded Functions

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
...
procedure Confused(I: Integer; J: Integer = 0); overload;
...
Confused(X);    // Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

## Default Parameters in Forward and Interface Declarations

If a routine has a forward declaration or appears in the interface section of a unit, default parameter values if there are any must be specified in the forward or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the forward or interface declaration exactly.

# Calling Procedures and Functions

---

This topic covers the following items:

- Program control and routine parameters
- Open array constructors

## Program Control and Parameters

---

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called actual parameters, while the parameters in the routine's declaration are called formal parameters.

When calling a routine, remember that

- expressions used to pass typed const and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass var and out parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass var and out parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(, , X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();  
DoSomething;
```

## Open Array Constructors

---

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;  
    procedure Add(A: array of Integer);
```

you could call the [Add](#) procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;  
    ...  
    Temp[0] := 5;  
    Temp[1] := 7;  
    Temp[2] := I;  
    Temp[3] := I + J;  
    Add(Temp);
```

Open array constructors can be passed only as value or const parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

# Classes and Objects

---

This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.

# Classes and Objects

---

This topic covers the following material:

- Declaration syntax of classes
- Inheritance and scope
- Visibility of class members
- Forward declarations and mutually dependent classes

## Class Types

---

A class, or class type, defines a structure consisting of fields, methods, and properties. Instances of a class type are called objects. The fields, methods, and properties of a class are called its components or members.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class' fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects that are instances of a class. Some methods (called class methods) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have access specifiers, which determine how their data is read and modified. From other parts of a program outside of the object itself a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called constructors and destructors.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value `nil`. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the `Size` property of the object referenced by `SomeObject`; you would not write this as `SomeObject^.Size := 100`.

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type
  className = class (ancestorClass)
```



```

    memberList
end;

```

where *className* is any valid identifier, (*ancestorClass*) is optional, and *memberList* declares members - that is, fields, methods, and properties - of the class. If you omit (*ancestorClass*), then the new class inherits directly from the predefined TObject class. If you include (*ancestorClass*) and *memberList* is empty, you can omit end. A class type declaration can also include a list of interfaces implemented by the class; see Implementing Interfaces.

Delphi for .NET supports the additional features of sealed classes and abstract classes. A sealed class is one that cannot be extended through inheritance. This includes all .NET languages that might use the sealed class. Delphi for .NET also allows an entire class to be declared as abstract, even though it does not contain any abstract virtual methods. The class declaration syntax for Delphi for .NET is:

```

type
  className = class [abstract | sealed] (ancestorType)
    memberList
  end;

```

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the TMemoryStream class from the [Classes](#) unit.

```

type TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint;
  override;
end;

```

TMemoryStream descends from TCustomMemoryStream (in the [Classes](#) unit), inheriting most of its members. But it defines - or redefines - several methods and properties, including its destructor method, [Destroy](#). Its constructor, [Create](#), is inherited without change from TObject, and so is not redeclared. Each member is declared as private, protected, or public (this class has no published members). These terms are explained below.

Given this declaration, you can create an instance of `TMemoryStream` as follows:

```
var stream: TMemoryStream;  
    stream := TMemoryStream.Create;
```

## Inheritance and Scope

---

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called `TSomeControl` that descends from `TControl`. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence `TSomeControl` contains all of the members defined in `TControl` and in each of `TControl`'s ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

## TObject and TClass

The `TObject` class, declared in the `System` unit, is the ultimate ancestor of all other classes. `TObject` defines only a handful of methods, including a basic constructor and destructor. In addition to `TObject`, the `System` unit declares the class reference type `TClass`:

```
TClass = class of TObject;
```

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from `TObject`. Thus

```
type TMyClass = class  
    ...  
end;
```

is equivalent to

```
type TMyClass = class(TObject)  
    ...  
end;
```

The latter form is recommended for readability.

## Compatibility of Class Types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
    TFigure = class(TObject);
    TRectangle = class(TFigure);
    TSquare = class(TRectangle);
var
    Fig: TFigure;
```

the variable `Fig` can be assigned values of type `TFigure`, `TRectangle`, and `TSquare`.

## Object Types

The Win32 Delphi compiler allows an alternative syntax to class types, which you can declare object types using the syntax

```
type objectType = object (ancestorObjectType)
    memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from `TObject`, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the `New` procedure and destroy them with the `Dispose` procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended on Win32, and they have been completely deprecated in the Delphi for .NET compiler.

## Visibility of Class Members

---

Every member of a class has an attribute called visibility, which is indicated by one of the reserved words `private`, `protected`, `public`, `published`, or `automated`. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called `Color`. Visibility determines where and how a member can be accessed, with `private` representing the least accessibility, `protected` representing an intermediate level of accessibility, and `public`, `published`, and `automated` representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default published, provided the class is compiled in the `{ $M+ }` state or is derived from a class compiled in the `{ $M+ }` state; otherwise, such members are public.

For readability, it is best to organize a class declaration by visibility, placing all the private members together, followed by all the protected members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new 'section' of the declaration. So a typical class declaration should like this:

```
type
  TMyClass = class(TControl)
    private
      ... { private declarations here }
    protected
      ... { protected declarations here }
    public
      ... { public declarations here }
    published
      ... { published declarations here }
  end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a protected property can be made public in a descendant, but not private. Moreover, published members cannot become public in a descendant class. For more information, see [Property overrides and redeclarations](#).

## Private, Protected, and Public Members

A private member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another's private members without making those members more widely accessible.

A protected member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. A protected method can be called, and a protected field or property read or written to, from the definition of any method belonging to a class that descends from the one where the protected member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A public member is visible wherever its class can be referenced.

## Additional Visibility Specifiers for .NET

In addition to private and protected visibility specifiers, the Delphi for .NET compiler supports additional visibility settings that comply with the .NET Common Language Specification (CLS). These are, strict private, and strict protected visibility.

Class members with strict private visibility are accessible only within the class in which they are declared. They are not visible to procedures or functions declared within the same unit.

Class members with strict protected visibility are visible within the class in which they are declared, and within any descendant class, regardless of where it is declared.

Delphi's traditional private visibility specifier maps to the CLR's `assembly` visibility. Delphi's protected visibility specifier maps to the CLR's `assembly` or `family` visibility.

**Note:** The word *strict* is treated as a directive within the context of a class declaration. Within a class declaration you cannot declare a member named 'strict', but it is acceptable for use outside of a class declaration.

## Published Members

Published members have the same visibility as public members. The difference is that runtime type information (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the *Object Inspector*, and to associate specific methods (called event handlers) with specific properties (called events).

Published properties are restricted to certain data types. Ordinal, string, class, interface, variant, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except Real48 can be published. Properties of an array type (as distinct from array properties, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types, and properties of enumerated types that include anonymous values. If you publish a property of this kind, the *Object Inspector* won't display it correctly, nor will the property's value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the `{ $M+ }` state or descends from a class compiled in the `{ $M+ }` state. Most classes with published members derive from `TPersistent`, which is compiled in the `{ $M+ }` state, so it is seldom necessary to use the `$M` directive.

## Automated Members (Win32 Only)

Automated members have the same visibility as public members. The difference is that Automation type information (required for Automation servers) is generated for automated members. Automated

members typically appear only in Win32 classes, and the automated reserved word has been deprecated in the .NET compiler. The automated reserved word is maintained for backward compatibility. The TAutoObject class in the ComObj unit does not use automated.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, and all interface types.
- Method declarations must use the default register calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (read and write) but other specifiers (index, stored, default, and nodefault) are not allowed. Access specifiers must list a method identifier that uses the default register calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a dispid directive. Specifying an already used ID in a dispid directive causes an error.

On the Win32 platform, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Win32 only), see Automation objects.

## Forward Declarations and Mutually Dependent Classes

---

If the declaration of a class type ends with the word `class` and a semicolon - that is, if it has the form

type *className* = class;

with no ancestor or class members listed after the word `class`, then it is a forward declaration. A forward declaration must be resolved by a defining declaration of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
    TFigure = class;    // forward declaration
    TDrawing = class
        Figure: TFigure;
        ...
    end;

    TFigure = class    // defining declaration
```

```
        Drawing: TDrawing;  
        ...  
    end;
```

Do not confuse forward declarations with complete declarations of types that derive from TObject without declaring any class members.

```
type  
    TFirstClass = class;    // this is a forward declaration  
    TSecondClass = class    // this is a complete class declaration  
    end;                   //  
    TThirdClass = class(TObject); // this is a complete class declaration
```

# Fields

---

This topic describes the syntax of class data fields declarations.

## About Fields

---

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any property or method declarations. For example, the following declaration creates a class called `TNumber` whose only member, other than the methods it inherits from `TObject`, is an integer field called `Int`.

```
type TNumber = class
    Int: Integer;
end;
```

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
    TAncestor = class
        Value: Integer;
    end;

    TDescendant = class(TAncestor)
        Value: string;    // hides the inherited Value field
    end;

var
    MyObject: TAncestor;

begin
    MyObject := TDescendant.Create;
    MyObject.Value := 'Hello!'    // error

    (MyObject as TDescendant).Value := 'Hello!'    // works!
end;
```

Although `MyObject` holds an instance of `TDescendant`, it is declared as `TAncestor`. The compiler therefore interprets `MyObject.Value` as referring to the (integer) field declared in `TAncestor`. Both fields, however, exist in the `TDescendant` object; the inherited `Value` is hidden by the new one, and can be accessed through a typecast.



## Class Fields (.NET)

---

Class fields are data fields in a class that can be accessed without an object reference.

You can introduce a block of class fields within a class declaration by using the class var block declaration. All fields declared after class var have static storage attributes. A class var block is terminated by the following:

1. Another class var declaration
2. A procedure or function (i.e. method) declaration (including class procedures and class functions)
3. A property declaration (including class properties)
4. A constructor or destructor declaration
5. A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```
type
  TMyClass = class
    public
      class var          // Introduce a block of class static fields.
        Red: Integer;
        Green: Integer;
        Blue: Integer;
      procedure Proc1; // Ends the class var block.
    end;
```

The above class fields can be accessed with the code:

```
TMyClass.Red := 0;
TMyClass.Green := 0;
TMyClass.Blue := 0;
```

# Methods

---

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example,

`SomeObject.Free` calls the `Free` method in `SomeObject`.

This topic covers the following material:

- Methods declarations and implementation
- Method binding
- Overloading methods
- Constructors and destructors
- Message methods

## About Methods

---

Within a class declaration, methods appear as procedure and function headings, which work like forward declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of `TMyClass` includes a method called `DoSomething`:

```
type
  TMyClass = class(TObject)
    ...
    procedure DoSomething;
    ...
  end;
```

A defining declaration for `DoSomething` must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
  ...
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class' methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does, the order, type and names of the parameters must match exactly, and if the method is a function, the return value must match as well.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

reintroduce; overload; *binding*; *calling convention*; abstract; *warning*

where *binding* is virtual, dynamic, or override; calling convention is register, pascal, cdecl, stdcall, or safecall; and *warning* is platform, deprecated, or library.

## Inherited

The reserved word inherited plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If inherited is followed by the name of a member, it represents a normal method call or reference to a property or field - except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited [Create](#).

When inherited has no identifier after it, it refers to the inherited method with the same name as the enclosing method or, if the enclosing method is a message handler, to the inherited message handler for the same message. In this case, inherited takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

## Self

Within the implementation of a method, the identifier Self references the object in which the method is called. For example, here is the implementation of TCollection's [Add](#) method in the [Classes](#) unit.

```
function TCollection.Add: TCollectionItem;  
begin  
    Result := FItemClass.Create(Self);  
end;
```

The [Add](#) method calls the [Create](#) method in the class referenced by the [FItemClass](#) field, which is always a TCollectionItem descendant. [TCollectionItem.Create](#) takes a single parameter of type TCollection, so [Add](#) passes it the TCollection instance object where [Add](#) is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
```

```

...
    MyCollection.Add    // MyCollection is passed to the TCollectionItem.
Create method

```

Self is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class' methods. In this case, you can access the original member identifier as `Self.Identifier`.

For information about Self in class methods, see [Class methods](#).

## Method Binding

---

Method bindings can be static (the default), virtual, or dynamic. Virtual and dynamic methods can be overridden, and they can be abstract. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

## Static Methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the `Draw` methods are static.

```

type
    TFigure = class
        procedure Draw;
    end;

    TRectangle = class(TFigure)
        procedure Draw;
    end;

```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to `Figure.Draw`, the `Figure` variable references an object of class `TRectangle`, but the call invokes the implementation of `Draw` in `TFigure`, because the declared type of the `Figure` variable is `TFigure`.

```

var
    Figure: TFigure;
    Rectangle: TRectangle;

begin
    Figure := TFigure.Create;
    Figure.Draw;           // calls TFigure.Draw
    Figure.Destroy;
    Figure := TRectangle.Create;

```

```

    Figure.Draw;                // calls TFigure.Draw

    TRectangle(Figure).Draw;    // calls TRectangle.Draw

    Figure.Destroy;
    Rectangle := TRectangle.Create;
    Rectangle.Draw;             // calls TRectangle.Draw
    Rectangle.Destroy;
end;

```

## Virtual and Dynamic Methods

To make a method virtual or dynamic, include the virtual or dynamic directive in its declaration. Virtual and dynamic methods, unlike static methods, can be overridden in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call not the declared type of the variable determines which implementation to activate.

To override a method, redeclare it with the override directive. An override declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the `Draw` method declared in `TFigure` is overridden in two descendant classes.

```

type
  TFigure = class
    procedure Draw; virtual;
  end;

  TRectangle = class(TFigure)
    procedure Draw; override;
  end;

  TEllipse = class(TFigure)
    procedure Draw; override;
  end;

```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```

var
  Figure: TFigure;

begin
  Figure := TRectangle.Create;
  Figure.Draw;      // calls TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;

```

```

    Figure.Draw;          // calls TEllipse.Draw
    Figure.Destroy;
end;

```

Only virtual and dynamic methods can be overridden. All methods, however, can be overloaded; see [Overloading methods](#).

The Delphi for .NET compiler supports the concept of a *final* virtual method. When the keyword *final* is applied to a virtual method, no ancestor class can override that method. Use of the *final* keyword is an important design decision that can help document how the class is intended to be used. It can also give the .NET JIT compiler hints that allow it to optimize the code it produces.

## Virtual Versus Dynamic

Virtual and dynamic methods are semantically equivalent. They differ only in the implementation of method-call dispatching at runtime. Virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

**Note:** Only use dynamic methods if there is a clear, observable benefit. Generally, use virtual methods.

## Overriding Versus Hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include *override*, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```

type
  T1 = class(TObject)
    procedure Act; virtual;
  end;

  T2 = class(T1)
    procedure Act;    // Act is redeclared, but not overridden
  end;

var
  SomeObject: T1;

begin
  SomeObject := T2.Create;
  SomeObject.Act;    // calls T1.Act
end;

```

## Reintroduce

The reintroduce directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```
procedure DoSomething; reintroduce;    // the ancestor class also has a
DoSomething method
```

Use reintroduce when you want to hide an inherited virtual method with a new one.

## Abstract Methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive abstract after virtual or dynamic. For example,

```
procedure DoSomething; virtual; abstract;
```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

**Note:** The Delphi for .NET compiler allows an entire class to be declared abstract, even though it does not contain any virtual abstract methods. See Class Types for more information.

## Class Static Methods (.NET)

Like class fields and class properties, class static methods can be accessed without an object reference. Unlike Win32 Delphi class methods, class static methods have no Self parameter at all, and therefore cannot access any class members. Also unlike Win32 Delphi, class static methods cannot be declared virtual.

Methods are made class static by appending the word static to their declaration, for example

```
type
  TMyClass = class
    strict private
      class var
        FX: Integer;

    strict protected

    // Note: accessors for class properties must be declared class static.
    function GetX: Integer; static;
    procedure SetX(val: Integer); static;

  public
    class property X: Integer read GetX write SetX;
    procedure StatProc(s: String); static;
```

```
end;
```

You can call a class static method through the class type (i.e. without having an object reference), for example

```
TMyClass.X := 17;  
TMyClass.StatProc('Hello');
```

## Overloading Methods

---

A method can be redeclared using the overload directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the reintroduce directive when you redeclare it in descendant classes. For example,

```
type  
  T1 = class(TObject)  
    procedure Test(I: Integer); overload; virtual;  
  end;  
  
  T2 = class(T1)  
    procedure Test(S: string); reintroduce; overload;  
  end;  
  ...  
  
SomeObject := T2.Create;  
SomeObject.Test('Hello!');           // calls T2.Test  
SomeObject.Test(7);                  // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type  
  TSomeClass = class  
    published  
      function Func(P: Integer): Integer;  
      function Func(P: Boolean): Integer;    // error  
    ...
```

Methods that serve as property read or write specifiers cannot be overloaded.



The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see [Overloading procedures and functions](#).

## Constructors

---

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word `constructor`. Examples:

```
constructor Create;  
constructor Create(AOwner: TComponent);
```

Constructors must use the default register calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor `Create`.

To create an object, call the constructor method on a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object, sets the values of all ordinal fields to zero, assigns nil to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the `Destroy` destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor's implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word `inherited` to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type  
  TShape = class(TGraphicControl)  
    private  
      FPen: TPen;  
      FBrush: TBrush;  
      procedure PenChanged(Sender: TObject);  
      procedure BrushChanged(Sender: TObject);
```

```

        public
            constructor Create(Owner: TComponent); override;
            destructor Destroy; override;
            ...
        end;

constructor TShape.Create(Owner: TComponent);
begin
    inherited Create(Owner);      // Initialize inherited parts
    Width := 65;                  // Change inherited properties
    Height := 65;
    FPen := TPen.Create; // Initialize new fields
    FPen.OnChange := PenChanged;
    FBrush := TBrush.Create;
    FBrush.OnChange := BrushChanged;
end;

```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), nil (pointer and class types), empty (string types), or Unassigned (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as virtual is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects that is, construction of objects whose types aren't known at compile time. (See Class references.)

**Note:** For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic [Memory Management Issues on the .NET Platform](#).

## The Class Constructor (.NET)

---

A class constructor executes before a class is referenced or used. The class constructor must be declared as strict private, and there can be at most one class constructor declared in a class. Descendants can declare their own class constructor, however, do not call inherited within the body of a class constructor. In fact, you cannot call a class constructor directly, or access it in any way (such as taking its address). The compiler generates code to call class constructors for you.

There can be no guarantees on when a class constructor will execute, except to say that it will execute at some time before the class is used. On the .NET platform in order for a class to be "used", it must reside in code that is actually executed. For example, if a class is first referenced in an if statement, and the test of the if statement is never true during the course of execution, then the class will never be loaded and JIT compiled. Hence, in this case the class constructor would not be called.

The following class declaration demonstrates the syntax of class properties and fields, as well as class static methods and class constructors:

```

type
  TMyClass = class
    strict protected

        // Accessors for class properties must be declared class static.
        class function GetX: Integer; static;
        class procedure SetX(val: Integer); static;
    public
        class property X: Integer read GetX write SetX;
        class procedure StatProc(s: String); static;
    strict private
        class var
            FX: Integer;
        class constructor Create;
end;

```

## Destructors

---

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word `destructor`. Example:

```

destructor SpecialDestructor(SaveData: Boolean);
destructor Destroy; override;

```

Destructors on Win32 must use the default register calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited `Destroy` method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```

MyObject.Destroy;

```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```

destructor TShape.Destroy;
begin
    FBrush.Free;
    FPen.Free;
    inherited Destroy;
end;

```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, `Destroy` is automatically called to dispose of the unfinished object. This means that `Destroy` must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always nil. A destructor should therefore check for nil values before operating on class-type or pointer-type fields. Calling the `Free` method (defined in `TObject`), rather than `Destroy`, offers a convenient way of checking for nil values before destroying an object.

**Note:** For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic [Memory Management Issues on the .NET Platform](#).

## Message Methods

---

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. VCL uses message methods to respond to Windows messages.

A message method is created by including the message directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Win32 message IDs defined, along with corresponding record types, in the [Messages](#) unit. A message method must be a procedure that takes a single var parameter.

For example:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

A message method does not have to include the `override` directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

## Implementing Message Methods

The implementation of a message method can call the inherited message method, as in this example:

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
```

```

    if Message.CharCode = Ord(#13) then
        ProcessEnter
    else
        inherited;
    end;
end;

```

The `inherited` statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, `inherited` calls the `DefaultHandler` method originally defined in `TObject`.

The implementation of `DefaultHandler` in `TObject` simply returns without performing any actions. By overriding `DefaultHandler`, a class can implement its own default handling of messages. On Win32, the `DefaultHandler` method for controls calls the Win32 API `DefWindowProc`.

## Message Dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the `Dispatch` method inherited from `TObject`:

```

procedure Dispatch(var Message);

```

The `Message` parameter passed to `Dispatch` must be a record whose first entry is a field of type `Word` containing a message ID.

`Dispatch` searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, `Dispatch` calls `DefaultHandler`.

# Properties

---

This topic describes the following material:

- Property access
- Array properties
- Index specifiers
- Storage specifiers
- Property overrides and redeclarations
- Class properties (.NET)

## About Properties

---

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

```
property propertyName[indexes]: type index integerConstant specifiers;
```

where

- *propertyName* is any valid identifier.
- [*indexes*] is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier1*, ..., *identiern*: type. For more information, see Array Properties, below.
- type must be a predefined or previously declared type identifier. That is, property declarations like `property Num: 0..9 ...` are invalid.
- the index *integerConstant* clause is optional. For more information, see Index Specifiers, below.
- *specifiers* is a sequence of read, write, stored, default (or no default), and implements specifiers. Every property declaration must have at least one read or write specifier.

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as var parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a read method that retrieves a value from a database or generates a random value.

## Property Access

---

Every property has a read specifier, a write specifier, or both. These are called access specifiers and they have the form

read *fieldOrMethod*

write *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.
- If *fieldOrMethod* is a method, it cannot be dynamic and, if virtual, cannot be overloaded. Moreover, access methods for a published property must use the default register calling convention.
- In a read specifier, if *fieldOrMethod* is a method, it must be a parameterless function whose result type is the same as the property's type. (An exception is the access method for an indexed property or an array property.)
- In a write specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or const parameter of the same type as the property (or more, if it is an array property or indexed property).

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the `GetColor` method must be declared as

```
function GetColor: TColor;
```

and the `SetColor` method must be declared as one of these:

```
procedure SetColor(Value: TColor);  
procedure SetColor(const Value: TColor);
```

(The name of `SetColor`'s parameter, of course, doesn't have to be `Value`.)

When a property is referenced in an expression, its value is read using the field or method listed in the read specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the write specifier.

The example below declares a class called `TCompass` with a published property called `Heading`. The value of `Heading` is read through the `FHeading` field and written through the `SetHeading` procedure.

```
type  
  THeading = 0..359;  
  TCompass = class(TControl)  
    private  
      FHeading: THeading;
```

```

        procedure SetHeading(Value: THeading);
published
        property Heading: THeading read FHeading write SetHeading;
        ...
    end;

```

Given this declaration, the statements

```

if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;

```

correspond to

```

if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);

```

In the `TCompass` class, no action is associated with reading the `Heading` property; the read operation consists of retrieving the value stored in the `FHeading` field. On the other hand, assigning a value to the `Heading` property translates into a call to the `SetHeading` method, which, presumably, stores the new value in the `FHeading` field as well as performing other actions. For example, `SetHeading` might be implemented like this:

```

procedure TCompass.SetHeading(Value: THeading);
begin
    if FHeading <> Value then
    begin
        FHeading := Value;
        Repaint;    // update user interface to reflect new value
    end;
end;

```

A property whose declaration includes only a read specifier is a read-only property, and one whose declaration includes only a write specifier is a write-only property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

## Array Properties

---

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```

property Objects[Index: Integer]: TObject read GetObject write SetObject;

```



```
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a read specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a write specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or const parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\BIN');
```

The definition of an array property can be followed by the default directive, in which case the array property becomes the default property of the class. For example,

```
type
  TStringArray = class
    public
      property Strings[Index: Integer]: string ...; default;
      ...
  end;
```

If a class has a default property, you can access that property with the abbreviation `object[index]`, which is equivalent to `object.property[index]`. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always binds to properties statically.

## Index Specifiers

---

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive `index` followed by an integer constant between -2147483647 and 2147483647. If a property has an index specifier, its read and write specifiers must list methods rather than fields. For example,

```
type
  TRectangle = class
    private
      FCoordinates: array[0..3] of Longint;
      function GetCoordinate(Index: Integer): Longint;
      procedure SetCoordinate(Index: Integer; Value: Longint);
    public
      property Left: Longint index 0 read GetCoordinate write
SetCoordinate;
      property Top: Longint index 1 read GetCoordinate write SetCoordinate;
      property Right: Longint index 2 read GetCoordinate write
SetCoordinate;
      property Bottom: Longint index 3 read GetCoordinate write
SetCoordinate;
      property Coordinates[Index: Integer]: Longint read GetCoordinate
write SetCoordinate;
      ...
    end;
```

An access method for a property with an index specifier must take an extra value parameter of type `Integer`. For a read function, it must be the last parameter; for a write procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if `Rectangle` is of type `TRectangle`, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

## Storage Specifiers

---

The optional `stored`, `default`, and `nodefault` directives are called storage specifiers. They have no effect on program behavior, but control whether or not to save the values of published properties in form files.

The `stored` directive must be followed by `True`, `False`, the name of a Boolean field, or the name of a parameterless method that returns a Boolean value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no `stored` directive, it is treated as if `stored True` were specified.

The `default` directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited default value without specifying a new one, use the `nodefault` directive. The `default` and `nodefault` directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without `default` or `nodefault`, it is treated as if `nodefault` were specified. For reals, pointers, and strings, there is an implicit default value of 0, nil, and '' (the empty string), respectively.

**Note:** You can't use the ordinal value 2147483648 as a default value. This value is used internally to represent `nodefault`.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its default value (or if there is no default value) and the `stored` specifier is `True`, then the property's value is saved. Otherwise, the property's value is not saved.

**Note:** Property values are not automatically initialized to the default value. That is, the `default` directive controls only when property values are saved to the form file, but not the initial value of the property on a newly created instance.

Storage specifiers are not supported for array properties. The `default` directive has a different meaning when used in an array property declaration. See [Array Properties](#), above.

## Property Overrides and Redeclarations

---

A property declaration that doesn't specify a type is called a property override. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word `property` followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as `protected`, a derived class can redeclare it in a `public` or `published` section of the class. Property overrides can include `read`, `write`, `stored`, `default`, and `nodefault` directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an `implements` directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
  TAncestor = class
    ...
    protected
      property Size: Integer read FSize;
      property Text: string read GetText write SetText;
      property Color: TColor read FColor write SetColor stored False;
    ...
  end;

type
  TDerived = class(TAncestor)
    ...
    protected
      property Size write SetSize;
    published
      property Text;
      property Color stored True default clBlue;
    ...
  end;
```

The override of `Size` adds a write specifier to allow the property to be modified. The overrides of `Text` and `Color` change the visibility of the properties from protected to published. The property override of `Color` also specifies that the property should be filed if its value isn't `clBlue`.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always static. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to `MyObject.Value` invokes `Method1` or `Method2`, even though `MyObject` holds an instance of `TDescendant`. But you can cast `MyObject` to `TDescendant` to access the descendant class's properties and their access specifiers.

```
type
  TAncestor = class
    ...
    property Value: Integer read Method1 write Method2;
  end;
```

```

TDescendant = class(TAncestor)
...
property Value: Integer read Method3 write Method4;
end;

var MyObject: TAncestor;
...
MyObject := TDescendant.Create;

```

## Class Properties (.NET)

---

Class properties can be accessed without an object reference. Class property accessors must themselves be declared as class static methods, or class fields. A class property cannot be published, and cannot have stored or default value definitions.

You can introduce a block of class static properties within a class declaration by using the class var block declaration. All properties declared after class var have static storage attributes. A class var block is terminated by the following:

1. Another class var declaration
2. A procedure or function (i.e. method) declaration (including class procedures and class functions)
3. A property declaration (including class properties)
4. A constructor or destructor declaration
5. A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```

type
  TMyClass = class
    strict private
      class var          // Note fields must be declared as class fields
        FRed: Integer;
        FGreen: Integer;
        FBlue: Integer;
    public
      class var          // Introduce a block of class properties
        property Red: Integer read FRed write FRed;
        Green: Integer read FGreen write FGreen;
        Blue: Integer read FBlue write FBlue;
        procedure Proc1; // Ends the class var block.
  end;

```

You can access the above class properites with the code:

```

TMyClass.Red := 0;

```

```
TMyClass.Blue := 0;  
TMyClass.Green := 0;
```

# Class References

---

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

This topic covers the following material:

- Class reference types
- Class operators
- Class methods

## Class-Reference Types

---

A class-reference type, sometimes called a metaclass, is denoted by a construction of the form

`class of type`

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of type`. If `type1` is an ancestor of `type2`, then `class of type2` is assignment-compatible with class of `type1`. Thus

```
type TClass = class of TObjet;  
var AnyObj: TClass;
```

declares a variable called `AnyObj` that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value `nil` to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for `TCollection` (in the `Classes` unit):

```
type TCollectionItemClass = class of TCollectionItem;  
...  
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a `TCollection` instance object, you must pass to the constructor the name of a class descending from `TCollectionItem`.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

## Constructors and Class References

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
const ControlName: string; X, Y, W, H: Integer): TControl;
begin
    Result := ControlClass.Create(MainForm);
    with Result do
        begin
            Parent := MainForm;
            Name := ControlName;
            SetBounds(X, Y, W, H);
            Visible := True;
        end;
    end;
end;
```

The `CreateControl` function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to `CreateControl` can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

## Class Operators

---

Every class inherits from `TObject` methods called `ClassType` and `ClassParent` that return, respectively, a reference to the class of an object and of an object's immediate ancestor. Both methods return a value of type `TClass` (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called `InheritsFrom` that tests whether the object where it is called descends from a specified class. These methods are used by the `is` and `as` operators, and it is seldom necessary to call them directly.

### The `is` Operator

The `is` operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression



### *objectisclass*

returns True if *object* is an instance of the class denoted by *class* or one of its descendants, and False otherwise. (If *object* is nil, the result is False.) If the declared type of *object* is unrelated to *class* - that is, if the types are distinct and one is not an ancestor of the other a compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to `TEdit` after first verifying that the object it references is an instance of `TEdit` or one of its descendants.

## The as Operator

The as operator performs checked typecasts. The expression

### *objectasclass*

returns a reference to the same object as *object*, but with the type given by *class*. At runtime, *object* must be an instance of the class denoted by *class* or one of its descendants, or be nil; otherwise an exception is raised. If the declared type of *object* is unrelated to *class* - that is, if the types are distinct and one is not an ancestor of the other - a compilation error results. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

The rules of operator precedence often require as typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

## Class Methods

---

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word `class`. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ...
  end;
```

The defining declaration of a class method must also begin with `class`. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);  
begin  
    ...  
end;
```

In the defining declaration of a class method, the identifier `Self` represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class `C`, then `Self` is of the type class of `C`. Thus you cannot use `Self` to access fields, properties, and normal (object) methods, but you can use it to call constructors and other class methods.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of `Self`.

# Exceptions

---

This topic covers the following material:

- A conceptual overview of exceptions and exception handling
- Declaring exception types
- Raising and handling exceptions

## About Exceptions

---

An exception is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an exception handler, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the `SysUtils` unit, most runtime errors are automatically converted into exceptions. Many errors that would otherwise terminate an application - such as insufficient memory, division by zero, and general protection faults - can be caught and handled.

## When To Use Exceptions

---

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The requirements imposed by exception handling semantics impose a code/data size and runtime performance penalty. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a `try...except` or `try...finally` statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in `if...then` statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
  AssignFile(F, FileName);
  Reset(F);      // raises an EInOutError exception if file is not found
except
```

```

        on Exception do ...
    end;

```

But you could also avoid the overhead of exception handling by using

```

if FileExists(FileName) then      // returns False if file is not found;
    raises no exception

```

```

begin
    AssignFile(F, FileName);
    Reset(F);
end;

```

*Assertions* provide another way of testing a Boolean condition anywhere in your source code. When an `Assert` statement fails, the program either halts with a runtime error or (if it uses the `SysUtils` unit) raises an `EAssertionFailed` exception. Assertions should be used only to test for conditions that you do not expect to occur.

## Declaring Exception Types

---

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the `Exception` class defined in `SysUtils`.

You can group exceptions into families using inheritance. For example, the following declarations in `SysUtils` define a family of exception types for math errors.

```

type
    EMathError = class(Exception);
    EInvalidOp = class(EMathError);
    EZeroDivide = class(EMathError);
    EOverflow = class(EMathError);
    EUnderflow = class(EMathError);

```

Given these declarations, you can define a single `EMathError` exception handler that also handles `EInvalidOp`, `EZeroDivide`, `EOverflow`, and `EUnderflow`.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```

type EInOutError = class(Exception)
    ErrorCode: Integer;
end;

```

## Raising and Handling Exceptions

---

To raise an exception object, use an instance of the exception class with a raise statement. For example,

```
raise EMathError.Create;
```

In general, the form of a raise statement is

```
raise object at address
```

where object and at address are both optional. When an address is specified, it can be any expression that evaluates to a pointer type, but is usually a pointer to a procedure or function. For example:

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

Use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is raised - that is, referenced in a raise statement - it is governed by special exception-handling logic. A raise statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose `try...except` block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an `ERangeError` exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
    Result := StrToInt(S);    // StrToInt is declared in SysUtils
    if (Result < Min) or (Result > Max) then
        raise ERangeError.CreateFmt('%d is not within the valid range of %d..%d', [Result, Min, Max]);
end;
```

Notice the `CreateFmt` method called in the raise statement. Exception and its descendants have special constructors that provide alternative ways to create exception messages and context IDs.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

**Note:** Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the `SysUtils` unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units - including `SysUtils` - are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program. Similarly, raising an exception in the finalization section of a unit may not lead to the intended result if `SysUtils` has already been finalized when the exception has been raised.

## Try...except Statements

Exceptions are handled within `try...except` statements. For example,

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide `Y` by `Z`, but calls a routine named `HandleZeroDivide` if an `EZeroDivide` exception is raised.

The syntax of a `try...except` statement is

`try statementsexceptexceptionBlockend`

where *statements* is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

*elsestatements*

An exception handler has the form

*onidentifier: typedostatement*

where *identifier* is optional (if included, identifier can be any valid identifier), *type* is a type used to represent exceptions, and *statement* is any statement.

A `try...except` statement executes the statements in the initial statements list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial statements list, either by a `raise` statement in the statements list or by a procedure or function called from the statements list, an attempt is made to 'handle' the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler 'matches' an exception just in case the type in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the statement in the `else` clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered `try...except` statement that has not yet exited. If no appropriate handler, `else` clause, or statement list is found there, the search propagates to the next-most-recently entered `try...`

`except` statement, and so forth. If the outermost `try...except` statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the `try...except` statement where the handling occurs, and control is transferred to the executed exception handler, else clause, or statement list. This process discards all procedure and function calls that occurred after entering the `try...except` statement where the exception is handled. The exception object is then automatically destroyed through a call to its `Destroy` destructor and control is passed to the statement following the `try...except` statement. (If a call to the `Exit`, `Break`, or `Continue` standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. `EMathError` appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
    ...
except
    on EZeroDivide do HandleZeroDivide;
    on EOverflow do HandleOverflow;
    on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows `on...do`. The scope of the identifier is limited to that statement. For example,

```
try
    ...
except
    on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an else clause, the else clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
    ...
except
    on EZeroDivide do HandleZeroDivide;
    on EOverflow do HandleOverflow;
    on EMathError do HandleMathError;
else
```

```

    HandleAllOthers;
end;

```

Here, the `else` clause handles any exception that isn't an `EMathError`.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```

try
    ...
except
    HandleException;
end;

```

Here, the `HandleException` routine handles any exception that occurs as a result of executing the statements between `try` and `except`.

## Re-raising Exceptions

When the reserved word `raise` occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then re-raise the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the `GetFileList` function allocates a `TStringList` object and fills it with file names matching a specified search path:

```

function GetFileList(const Path: string): TStringList;
var
    I: Integer;
    SearchRec: TSearchRec;
begin
    Result := TStringList.Create;
    try
        I := FindFirst(Path, 0, SearchRec);
        while I = 0 do
            begin
                Result.Add(SearchRec.Name);
                I := FindNext(SearchRec);
            end;
        except
            Result.Free;
            raise;
        end;
    end;
end;

```

`GetFileList` creates a `TStringList` object, then uses the `FindFirst` and `FindNext` functions (defined in `SysUtils`) to initialize it. If the initialization fails - for example because the search path is



invalid, or because there is not enough memory to fill in the string list - `GetFileList` needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a `try...except` statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

## Nested Exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the `Tan` function below.

```
type
  ETrigError = class(EMathError);
function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
    end;
  end;
end;
```

If an `EMathError` exception occurs during execution of `Tan`, the exception handler raises an `ETrigError`. Since `Tan` does not provide a handler for `ETrigError`, the exception propagates beyond the original exception handler, causing the `EMathError` exception to be destroyed. To the caller, it appears as if the `Tan` function has raised an `ETrigError` exception.

## Try...finally Statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a `try...finally` statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
  ... // process file F
finally
```

```
    CloseFile(F);  
end;
```

The syntax of a `try...finally` statement is

```
try statementList1 finally statementList2 end
```

where each *statementList* is a sequence of statements delimited by semicolons. The `try...finally` statement executes the statements in *statementList1* (the try clause). If *statementList1* finishes without raising exceptions, *statementList2* (the finally clause) is executed. If an exception is raised during execution of *statementList1*, control is transferred to *statementList2*; once *statementList2* finishes executing, the exception is re-raised. If a call to the `Exit`, `Break`, or `Continue` procedure causes control to leave *statementList1*, *statementList2* is automatically executed. Thus the finally clause is always executed, regardless of how the try clause terminates.

If an exception is raised but not handled in the finally clause, that exception is propagated out of the `try...finally` statement, and any exception already raised in the try clause is lost. The finally clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

## Standard Exception Classes and Routines

---

The `SysUtils` and `System` units declare several standard routines for handling exceptions, including `ExceptObject`, `ExceptAddr`, and `ShowException`. `SysUtils`, `System` and other units also include dozens of exception classes, all of which (aside from `OutlineError`) derive from `Exception`.

The `Exception` class has properties called `Message` and `HelpContext` that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways.

# Standard Routines and I/O

---

This section describes the standard routines included in the Delphi runtime library.

# Standard Routines and I/O

---

These topics discuss text and file I/O and summarize standard library routines. Many of the procedures and functions listed here are defined in the `System` and `SysInit` units, which are implicitly used with every application. Others are built into the compiler but are treated as if they were in the `System` unit.

Some standard routines are in units such as `SysUtils`, which must be listed in a `uses` clause to make them available in programs. You cannot, however, list `System` in a `uses` clause, nor should you modify the `System` unit or try to rebuild it explicitly.

## File Input and Output

---

The table below lists input and output routines.

### *Input and output procedures and functions*

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
ReadLn	Does what Read does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.

Rmdir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as Write, and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: typed, text, and untyped. The syntax for declaring file types is given in File types.

Before a file variable can be used, it must be associated with an external file through a call to the AssignFile procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be *opened* to prepare it for input or output. An existing file can be opened via the Reset procedure, and a new file can be created and opened via the Rewrite procedure. Text files opened with Reset are read-only and text files opened with Rewrite and Append are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with Reset or Rewrite.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure Read or written using the standard procedure Write, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure Seek, which moves the current file position to a specified component. The standard functions FilePos and FileSize can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure CloseFile. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{SI+}** and **{SI}** compiler directives. When I/O checking is off, that is, when a procedure or function call is compiled in the **{SI}** state an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function IOResult instead.

You must call the `IOResult` function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{!+}** is the current state, the next I/O function call will fail with the lingering `IOResult` error.

## Text Files

This section summarizes I/O using file variables of the standard type `Text`.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a line feed character). The type `Text` is distinct from the type file of `Char`.

For text files, there are special forms of `Read` and `Write` that let you read and write values that are not of type `Char`. Such values are automatically translated to and from their character representation. For example, `Read(F, I)`, where `I` is a type `Integer` variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in `I`.

There are two standard text file variables, `Input` and `Output`. The standard file variable `Input` is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable `Output` is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, `Input` and `Output` are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');  
Reset(Input);  
AssignFile(Output, '');  
Rewrite(Output);
```

**Note:** For Win32 applications, text-oriented I/O is available only in console applications, that is, applications compiled with the Generate console application option checked on the Linker page of the Project Options dialog box or with the `-cc` command-line compiler option. In a GUI (non-console) application, any attempt to read or write using `Input` or `Output` will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, `Input` or `Output` is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using `AssignFile`, and opened using `Reset`, `Rewrite`, or `Append`. An error occurs if you pass a file that was opened with `Reset` to an output-oriented procedure or function. An error also occurs if you pass a file that was opened with `Rewrite` or `Append` to an input-oriented procedure or function.

## Untyped Files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word `file` and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the `Reset` and `Rewrite` procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for `Read` and `Write`, all typed-file standard procedures and functions are also allowed on untyped files. Instead of `Read` and `Write`, two procedures called `BlockRead` and `BlockWrite` are used for high-speed data transfers.

## Text File Device Drivers

---

You can define your own text file device drivers for your programs. A text file device driver is a set of four functions that completely implement an interface between Delphi's file system and some device.

The four functions that define each device driver are `Open`, `InOut`, `Flush`, and `Close`. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where `DeviceFunc` is the name of the function (that is, `Open`, `InOut`, `Flush`, or `Close`). The return value of a device-interface function becomes the value returned by `IOResult`. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized `Assign` procedure. The `Assign` procedure must assign the addresses of the four device-interface functions to the four function pointers in the text file variable. In addition, it should store the `fmClosed` magic constant in the `Mode` field, store the size of the text file buffer in `BufSize`, store a pointer to the text file buffer in `BufPtr`, and clear the `Name` string.

Assuming, for example, that the four device-interface functions are called `DevOpen`, `DevInOut`, `DevFlush`, and `DevClose`, the `Assign` procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
```

```

    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
end;
end;

```

The device-interface functions can use the UserData field in the file record to store private information. This field isn't modified by the product file system at any time.

## The Open function

The Open function is called by the Reset, Rewrite, and Append standard procedures to open a text file associated with a device. On entry, the Mode field contains fmInput, fmOutput, or fmInOut to indicate whether the Open function was called from Reset, Rewrite, or Append.

The Open function prepares the file for input or output, according to the Mode value. If Mode specified fmInOut (indicating that Open was called from Append), it must be changed to fmOutput before Open returns.

Open is always called before any of the other device-interface functions. For that reason, AssignDev only initializes the OpenFunc field, leaving initialization of the remaining vectors up to Open. Based on Mode, Open can then install pointers to either input- or output-oriented functions. This saves the InOut, Flush functions and the CloseFile procedure from determining the current mode.

## The InOut function

The InOut function is called by the Read, Readln, Write, Writeln, Eof, Eoln, SeekEof, SeekEoln, and CloseFile standard routines whenever input or output from the device is required.

When Mode is fmInput, the InOut function reads up to BufSize characters into BufPtr<sup>^</sup>, and returns the number of characters read in BufEnd. In addition, it stores zero in BufPos. If the InOut function returns zero in BufEnd as a result of an input request, Eof becomes **True** for the file.

When Mode is fmOutput, the InOut function writes BufPos characters from BufPtr<sup>^</sup>, and returns zero in BufPos.

## The Flush function

The Flush function is called at the end of each Read, Readln, Write, and Writeln. It can optionally flush the text file buffer.

If Mode is fmInput, the Flush function can store zero in BufPos and BufEnd to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If Mode is fmOutput, the Flush function can write the contents of the buffer exactly like the InOut function, which ensures that text written to the device appears on the device immediately. If Flush does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.



## The Close function

The Close function is called by the CloseFile standard procedure to close a text file associated with a device. (The Reset, Rewrite, and Append procedures also call Close if the file they are opening is already open.) If Mode is fmOutput, then before calling Close, the file system calls the InOut function to ensure that all characters have been written to the device.

## Handling null-Terminated Strings

The Delphi language's extended syntax allows the Read, ReadLn, Str, and Val standard procedures to be applied to zero-based character arrays, and allows the Write, WriteLn, Val, AssignFile, and Rename standard procedures to be applied to both zero-based character arrays and character pointers.

## Null-Terminated String Functions

The following functions are provided for handling null-terminated strings.

### *Null-terminated string functions*

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using StrAlloc or StrNew.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using StrAlloc or StrNew.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.
StrMove	Moves a block of characters from one string to another.

StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with `Ansi-`. For example, the multibyte version of `StrPos` is `AnsiStrPos`. Multibyte character support is operating-system dependent and based on the current locale.

## Wide-Character Strings

The `System` unit provides three functions, `WideCharToString`, `WideCharLenToString`, and `StringToWideChar`, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

Assignment will also convert between strings. For instance, the following are both valid:

```
MyAnsiString := MyWideString;
MyWideString := MyAnsiString;
```

## Other Standard Routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines.

### *Other standard routines*

Procedure or function	Description
<code>Addr</code>	Returns a pointer to a specified object.
<code>AllocMem</code>	Allocates a memory block and initializes each byte to zero.
<code>ArcTan</code>	Calculates the arctangent of the given number.
<code>Assert</code>	Raises an exception if the passed expression does not evaluate to true.
<code>Assigned</code>	Tests for a <b>nil</b> (unassigned) pointer or procedural variable.
<code>Beep</code>	Generates a standard beep.
<code>Break</code>	Causes control to exit a <code>for</code> , <code>while</code> , or <code>repeat</code> statement.
<code>ByteToCharIndex</code>	Returns the position of the character containing a specified byte in a string.

Chr	Returns the character for a specified integer value.
Close	Closes a file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of for, while, or repeat statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type TDateTime to a string.
DateToStr	Converts a variable of type TDateTime to a string.
Dec	Decrements an ordinal variable or a typed pointer variable.
Dispose	Releases dynamically allocated variable memory.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	Uninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Releases allocated memory.
GetMem	Allocates dynamic memory and a pointer to the address of the block.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable or a typed pointer variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.

Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a dynamic allocated variable memory and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal integer value of an ordinal-type expression.
Pos	Returns the index of the first single-byte character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a value to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamically allocatable memory.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Converts an integer or real number into a string.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (TDateTime).
StrToDateTime	Converts a string to a TDateTime.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.

StrToTime	Converts a string to a time format (TDateTime).
StrUpper	Returns an ASCII string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type TDateTime to a string.
Trunc	Truncates a real number to an integer.
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarArrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.
VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.
VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

# Libraries and Packages

---

This section describes how to create static and dynamically loadable libraries in Delphi.

# Libraries and Packages

---

A dynamically loadable library is a dynamic-link library (DLL) on Win32, and an assembly (also a DLL) on the .NET platform. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

Delphi programs can call DLLs and assemblies written in other languages, and applications written in other languages can call DLLs or assemblies written in Delphi.

## Calling Dynamically Loadable Libraries

---

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in DLL or assembly, you must import them. This can be done in two ways: by declaring an external procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

The Delphi language does not support importing of variables from DLLs or assemblies.

## Static Loading

---

The simplest way to import a procedure or function is to declare it using the external directive. For example,

```
procedure DoSomething; external 'MYLIB.DLL';
```

If you include this declaration in a program, MYLIB.DLL is loaded once, when the program starts. Throughout execution of the program, the identifier `DoSomething` always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect external declarations into a separate "import unit" that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

## Dynamic Loading

---

You can access routines in a library through direct calls to Win32 APIs, including `LoadLibrary`, `FreeLibrary`, and `GetProcAddress`. These functions are declared in `Windows.pas`. On Linux, they are implemented for compatibility in `SysUtils.pas`; the actual Linux OS routines are `dlopen`, `dlclose`, and `dlsym` (all declared in `libc`; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example,

```
uses Windows, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  .
  .
  .
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
      end;
      FreeLibrary(Handle);
    end;
  end;
end;
```

When you import routines this way, the library is not loaded until the code containing the call to [LoadLibrary](#) executes. The library is later unloaded by the call to [FreeLibrary](#). This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.



# Writing Dynamically Loaded Libraries

---

The following topics describe elements of writing dynamically loadable libraries, including

- The exports clause.
- Library initialization code.
- Global variables.
- Libraries and system variables.

## Using Export Clause in Libraries

---

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word `library` (instead of `program`).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, `Min` and `Max`.

```
library MinMax;
    function Min(X, Y: Integer): Integer; stdcall;
begin
    if X < Y then Min := X else Min := Y;
end;
    function Max(X, Y: Integer): Integer; stdcall;
begin
    if X > Y then Max := X else Max := Y;
end;
    exports
        Min,
        Max;
begin
end.
```

If you want your library to be available to applications written in other languages, it's safest to specify `stdcall` in the declarations of exported functions. Other languages may not support Delphi's default register calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a `uses` clause, an `exports` clause, and the initialization code. For example,

```
library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;
exports
    InitEditors,
    DoneEditors name Done,
    InsertText name Insert,
    DeleteSelection name Delete,
```

```

    FormatSelection,
    PrintSelection name Print,
    .
    .
    .
    SetErrorHandler;
begin
    InitLibrary;
end.

```

You can put exports clauses in the interface or implementation section of a unit. Any library that includes such a unit in its uses clause automatically exports the routines listed the unit's exports clauses without the need for an exports clause of its own.

The directive `local`, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the `local` directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for stand-alone procedures and functions, but not for methods. A routine declared with `local` for example,

```
function Contraband(I: Integer): Integer; local;
```

does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

A routine is exported when it is listed in an exports clause, which has the form

```
exports entry1, ..., entryn;
```

where each entry consists of the name of a procedure, function, or variable (which must be declared prior to the exports clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional name specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive `resident`, which is maintained for backward compatibility and is ignored by the compiler.)

On the Win32 platform, an index specifier consists of the directive `index` followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no index specifier, the routine is automatically assigned a number in the export table.

**Note:** Use of index specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A name specifier consists of the directive `name` followed by a string constant. If an entry has no name specifier, the routine is exported under its original declared name, with the same spelling and case. Use a name clause when you want to export a routine under a different name. For example,

```
exports
DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the exports clause. For example,

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

On Win32, do not include index specifiers in entries for overloaded routines.

An exports clause can appear anywhere and any number of times in the declaration part of a program or library, or in the interface or implementation section of a unit. Programs seldom contain an exports clause.

## Library Initialization Code

---

The statements in a library's block constitute the library's initialization code. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an entry point procedure using the `DllProc` variable. The `DllProc` variable is similar to an exit procedure, which is described in Exit procedures; the entry point procedure executes when the library is loaded or unloaded.

Library initialization code can signal an error by setting the `ExitCode` variable to a nonzero value. `ExitCode` is declared in the `System` unit and defaults to zero, indicating successful initialization. If a library's initialization code sets `ExitCode` to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an entry point procedure.

```
library Test;
var
  SaveDllProc: Pointer;
  procedure LibExit(Reason: Integer);
begin
  if Reason = DLL_PROCESS_DETACH then
    begin
      .
      .          // library exit code
      .
    end;
    SaveDllProc(Reason);          // call saved entry point procedure
  end;
begin
  .
```

```

        .                // library initialization code
        .
        SaveDllProc := DllProc;           // save exit procedure chain
        DllProc := @LibExit;             // install LibExit exit procedure
    end.

```

`DllProc` is called when the library is first loaded into memory, when a thread starts or stops, or when the library is unloaded. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's entry point procedure.

## Global Variables in a Library

---

Global variables declared in a shared library cannot be imported by a Delphi application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries - or multiple instances of a library - to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

## Libraries and System Variables

---

Several variables declared in the `System` unit are of special interest to those programming libraries. Use `IsLibrary` to determine whether code is executing in an application or in a library; `IsLibrary` is always `False` in an application and `True` in a library. During a library's lifetime, `HInstance` contains its instance handle. `CmdLine` is always `nil` in a library.

The `DLLProc` variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. `DLLProc` is available on both Windows and Linux but its use differs on each. On Win32, `DLLProc` is used in multithreading applications.; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior.

To monitor operating-system calls, create a callback procedure that takes a single integer parameter for example,

```
procedure DLLHandler(Reason: Integer);
```

and assign the address of the procedure to the `DLLProc` variable. When the procedure is called, it passes to it one of the following values.

---

`DLL_PROCESS_DETACH` Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to `FreeLibrary`.

---

`DLL_PROCESS_ATTACH` Indicates that the library is attaching to the address space of the calling process as the result of a call to `LoadLibrary`.

---

`DLL_THREAD_ATTACH` Indicates that the current process is creating a new thread.

`DLL_THREAD_DETACH` Indicates that a thread is exiting cleanly.

---

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

## Exceptions and Runtime Errors in Libraries

---

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Delphi, the exception can be handled through a normal try...except statement.

On Win32, if the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code `$0EEDFADE`. The first entry in the `ExceptionInformation` array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Delphi exception object.

Generally, you should not let exceptions escape from your library. Delphi exceptions map to the OS exception model (including the .NET exception model)..

If a library does not use the `SysUtils` unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. Because the library has no way of knowing whether it was called from a Delphi program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

## Shared-Memory Manager (Win32 Only)

---

On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the `ShareMem` unit. The same is true if one application or DLL allocates memory with `New` or `GetMem` which is deallocated by a call to `Dispose` or `FreeMem` in another module. `ShareMem` should always be the first unit listed in any program or library uses clause where it occurs.

`ShareMem` is the interface unit for the `BORLANDMM.DLL` memory manager, which allows modules to share dynamically allocated memory. `BORLANDMM.DLL` must be deployed with applications and DLLs that use `ShareMem`. When an application or DLL uses `ShareMem`, its memory manager is replaced by the memory manager in `BORLANDMM.DLL`.

# Packages

---

The following topics describe packages and various issues involved in creating and compiling them.

- Package declarations and source files
- Naming packages
- The requires clause
- Avoiding circular package references
- Duplicate package references
- The contains clause
- Avoiding redundant source code uses
- Compiling packages
- Generated files
- Package-specific compiler directives
- Package-specific command-line compiler switches

## Understanding Packages

---

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange where code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

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 referencing runtime packages in their requires clauses.

On Win32, package files end with the .bpl (Borland package library) extension. On the .NET platform, packages are .NET assemblies, and end with an extension of .dll

Ordinarily, packages are loaded statically when an applications starts. But you can use the [LoadPackage](#) and [UnloadPackage](#) routines (in the [SysUtils](#) unit) to load packages dynamically.

**Note:** When an application utilizes packages, the name of each packaged unit still must appear in the uses clause of any source file that references it.

## Package Declarations and Source Files

---

Each package is declared in a separate source file, which should be saved with the .dpk extension to avoid confusion with other files containing Delphi code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains:

- a name for the package.
- a list of other packages required by the new package. These are packages to which the new package is linked.

- a list of unit files contained by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

```
package packageName;
requiresClause;
containsClause;
end.
```

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the `DATAx` package.

```
package DATAx;
  requires
    rtl,
    contains Db, DBLocal, DBXpress, ... ;
end.
```

The `requires` clause lists other, external packages used by the package being declared. It consists of the directive `requires`, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a `requires` clause.

The `contains` clause identifies the unit files to be compiled and bound into the package. It consists of the directive `contains`, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word `in` and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

**Note:** Thread-local variables (declared with `threadvar`) in a packaged unit cannot be accessed from clients that use the package.

## Naming packages

A compiled package involves several generated files. For example, the source file for the package called `DATAx` is `DATAx.DPK`, from which the compiler generates an executable and a binary image called `DATAx.BPL` (Win32) or `DATAx.DLL` (.NET), and `DATAx.DCP` (Win32) or `DATAx.DCPIL` (.NET). `DATAx` is used to refer to the package in the `requires` clauses of other packages, or when using the package in an application. Package names must be unique within a project.

## The `requires` clause

The `requires` clause lists other, external packages that are used by the current package. It functions like the `uses` clause in a unit file. An external package listed 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 a package make references to other packaged units, the other packages should be included in the first package's `requires` clause. If the other packages are omitted from the `requires` clause, the compiler loads the referenced units from their `.dcu` or `.dcuil` files.

## Avoiding circular package references

Packages cannot contain circular references in their `requires` clauses. 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.

## Duplicate package references

The compiler ignores duplicate references in a package's `requires` clause. For programming clarity and readability, however, duplicate references should be removed.

## The contains clause

The `contains` clause identifies the unit files to be bound into the package. Do not include file-name extensions in the `contains` clause.

## Avoiding redundant source code uses

A package cannot be listed in the `contains` clause of another package or the `uses` clause of a unit.

All units included directly in a package's `contains` clause, or indirectly in the `uses` clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in `requires` clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

## Compiling Packages

---

Packages are ordinarily compiled from the IDE using `.dpc` files generated by the *Project Manager*. You can also compile `.dpc` files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

## Generated Files

The following table lists the files produced by the successful compilation of a package.



## Compiled package files

File extension	Contents
DCP (Win32) or DCPILE (.NET)	A binary image containing a package header and the concatenation of all .dcu (Win32) or .dcuil (.NET) files in the package. A single .dcp or .dcuil file is created for each package. The base name for the file is the base name of the .dpc source file.
BPL (Win32) or DLL (.NET)	The runtime package. This file is a DLL on Win32 with special Borland-specific features. The base name for the package is the base name of the dpc source file.

## Package-Specific Compiler Directives

The following table lists package-specific compiler directives that can be inserted into source code.

### Package-specific compiler directives

Directive	Purpose
<code>{ \$IMPLICITBUILD OFF }</code>	Prevents a package from being implicitly recompiled later. Use in .dpc files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
<code>{ \$G- }</code> or <code>{ \$IMPORTEDDATA OFF }</code>	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.
<code>{ \$WEAKPACKAGEUNIT ON }</code>	Packages unit weakly.
<code>{ \$DENYPACKAGEUNIT ON }</code>	Prevents unit from being placed in a package.
<code>{ \$DESIGNONLY ON }</code>	Compiles the package for installation in the IDE. (Put in .dpc file.)
<code>{ \$RUNONLY ON }</code>	Compiles the package as runtime only. (Put in .dpc file.)

Including `{ $DENYPACKAGEUNIT ON }` in 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.

Other compiler directives may be included, if appropriate, in package source code.

## Package-Specific Command-Line Compiler Switches

The following package-specific switches are available for the command-line compiler.

### 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.

LE path	Specifies the directory where the compiled package file will be placed.
LN path	Specifies the directory where the package dcp or dcpil file will be placed.
LUpackageName [;packageName2;...]	Specifies additional runtime packages to use in an application. Used when compiling a project.
Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

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.

**Note:** When using the `-LU` switch on the .NET platform, you can refer to the package with or without the `.dll` extension. If you omit the `.dll` extension, the compiler will look for the package on the unit search path, and on the package search path. However, if the package specification contains a drive letter or the path separator character, then the compiler will assume the package name is the full file name (including the `.dll` extension). In the latter case, if you specify a full or relative path, but omit the `.dll` extension, the compiler will not be able to locate the package.

# Object Interfaces

---

This section describes the use of interfaces in Delphi.

# Object Interfaces

---

An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models (such as CORBA and SOAP). Using a distributed object model, custom objects that support interfaces can interact with objects written in C++, Java, and other languages.

## Interface Types

---

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
    [{GUID}]
    memberList
end;
```

where (*ancestorInterface*) and [*{GUID}*] are optional. In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property read and write specifiers must be methods.
- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as default.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as virtual, dynamic, abstract, or override. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
IMalloc = interface(IInterface)
    [{00000002-0000-0000-C000-000000000046}]
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
```

```

        function DidAlloc(P: Pointer): Integer; stdcall;
        procedure HeapMinimize; stdcall;
end;

```

In some interface declarations, the interface reserved word is replaced by `dispinterface`. This construction (along with the `dispid`, `read only`, and `write only` directives) is platform-specific and is not used in Linux programming.

## Interface and Inheritance

---

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not implement methods. What an interface inherits is the obligation to implement methods an obligation that is passed onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of `IInterface`, which is defined in the `System` unit and is the ultimate ancestor of all other interfaces. On Win32, `IInterface` declares three methods: `QueryInterface`, `_AddRef`, and `_Release`. These methods are not present on the .NET platform, and you do not need to implement them.

**Note:** `IInterface` is equivalent to `IUnknown`. You should generally use `IInterface` for platform independent applications and reserve the use of `IUnknown` for specific programs that include Win32 dependencies.

`QueryInterface` provides the means to obtain a reference to the different interfaces that an object supports. `_AddRef` and `_Release` provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the `System` unit's `TInterfacedObject`. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects, however, must be managed through `_AddRef` and `_Release`.

## Interface Identification

---

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

```
['{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}']
```

where each x is a hexadecimal digit (0 through 9 or A through F). The Type Library editor automatically generates GUIDs for new interfaces. You can also generate GUIDs by pressing Ctrl+Shift+G in the code editor.

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations.

The `TGUID` and `PGUID` types, declared in the `System` unit, are used to manipulate GUIDs.

```

type
    PGUID = ^TGUID;
    TGUID = packed record
        D1: Longword;
        D2: Word;
        D3: Word;
        D4: array[0..7] of Byte;
    end;
end;

```

On the .NET platform, you can tag an interface as described above (i.e. following the interface declaration). However, if you use the traditional Delphi syntax, the first square bracket construct following the interface declaration is taken as a GUID specifier - not as a .NET attribute. (Note that .NET attributes always apply to the *next* symbol, not the previous one.) You can also associate a GUID with an interface using the .NET Guid custom attribute. In this case you would use the .NET style syntax, placing the attribute immediately before the interface declaration.

GUIDs are not required for interfaces in the .NET framework. They are only used for COM interoperability.

When you declare a typed constant of type TGUID, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type TGUID. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways

```
if Supports(Allocator, IMalloc) then ...
```

or

```
if Supports(Allocator, IID_IMalloc) then ...
```

## Calling Conventions for Interfaces

---

The default calling convention for interface methods is register, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with stdcall. Use safecall to implement CORBA interfaces. On Win32, you can use safecall to implement methods of dual interfaces.

## Interface Properties

---

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled.

In an interface, property read and write specifiers must be methods, since fields are not available.

## Forward Declarations

---

An interface declaration that ends with the reserved word `interface` and a semicolon, without specifying an ancestor, GUID, or member list, is a forward declaration. A forward declaration must be resolved by a defining declaration of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    .
    .
    .
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    .
    .
    .
  end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive `IWindow` from `IControl` and also derive `IControl` from `IWindow`.

# Implementing Interfaces

---

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor.

## Class Declarations

---

Such declarations have the form

```
type className = class (ancestorClass, interfacer1, ..., interfacerN)
    memberList
end;
```

For example,

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    .
    .
    .
end;
```

declares a class called `TMemoryManager` that implements the `IMalloc` and `IErrorInfo` interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the (Win32) declaration of `TInterfacedObject` in the `System` unit. On the .NET platform, `TInterfacedObject` is an alias for `TObject`.

```
type
    TInterfacedObject = class(TObject, IInterface)
    protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
    public
        procedure AfterConstruction; override;
        procedure BeforeDestruction; override;
        class function NewInstance: TObject; override;
```



```

        property RefCount: Integer read FRefCount;
    end;

```

TInterfacedObject implements the IInterface interface. Hence TInterfacedObject declares and implements each of IInterface's three methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares TMemoryManager as a direct descendent of TInterfacedObject.) On the Win32 platform, every interface inherits from IInterface, and a class that implements interfaces must implement the QueryInterface, \_AddRef, and \_Release methods. The System unit's TInterfacedObject implements these methods and is thus a convenient base from which to derive other classes that implement interfaces. On the .NET platform, IInterface does not declare these methods, and you do not need to implement them.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

## Method Resolution Clause

---

You can override the default name-based mappings by including method resolution clauses in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```

procedure interface.interfaceMethod = implementingMethod;

```

or

```

function interface.interfaceMethod = implementingMethod;

```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```

type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
        function IMalloc.Alloc = Allocate;
        procedure IMalloc.Free = Deallocate;
        .
        .
        .
    end;

```

maps `IMalloc`'s `Alloc` and `Free` methods onto `TMemoryManager`'s `Allocate` and `Deallocate` methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

## Changing Inherited Implementations

---

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class' declaration. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    .
    .
    .
end;
TWindow = class(TInterfacedObject, IWindow) // TWindow implements IWindow
  procedure Draw;
  .
  .
  .
end;
TFrameWindow = class(TWindow, IWindow) // TFrameWindow reimplements IWindow
  procedure Draw;
  .
  .
  .
end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

## Implementing Interfaces by Delegation

---

The **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements
IMyInterface;
```

declares a property called `MyInterface` that implements the interface `IMyInterface`.

The implements directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a read specifier. If the property uses a read method, that method must use the default register calling convention and cannot be dynamic (though it can be virtual) or specify the message directive.

The class you use to implement the delegated interface should derive from [TAggregationObject](#).

## Delegating to an Interface-Type Property

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the implements directive, and which does so without method resolution clauses. For example,

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
TMyClass = class(TObject, IMyInterface)
  FMyInterface: IMyInterface;
  property MyInterface: IMyInterface read FMyInterface implements
IMyInterface;
end;
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...// some object whose class implements
IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

## Delegating to a Class-Type Property

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve

ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```

type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements
      IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  end;
  procedure TMyImplClass.P1;
  .
  .
  .
  procedure TMyImplClass.P2;
  .
  .
  .
  procedure TMyClass.MyP1;
  .
  .
  .
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;    // calls TMyClass.MyP1;
  MyInterface.P2;    // calls TImplClass.P2;
end;

```

# Interface References

---

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. These topics describe Interface references and related topics.

## Implementing Interface References

---

Interface reference variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following:

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
type
    IAncestor = interface
end;
IDescendant = interface(IAncestor)
    procedure P1;
end;
TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
end;
.
.
.
var
    D: IDescendant;
    A: IAncestor;
begin
    D := TSomething.Create; // works!
    A := TSomething.Create; // error
    D.P1; // works!
    D.P2; // error
end;
```

In this example, A is declared as a variable of type `IAncestor`. Because `TSomething` does not list `IAncestor` among the interfaces it implements, a `TSomething` instance cannot be assigned to A. But if we changed `TSomething`'s declaration to

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
.
```

.  
.  
the first error would become a valid assignment. D is declared as a variable of type `IDescendant`. While D references an instance of `TSomething`, we cannot use it to access `TSomething`'s P2 method, since P2 is not a method of `IDescendant`. But if we changed D's declaration to

```
D: TSomething;
```

the second error would become a valid method call.

On the Win32 platform, interface references are typically managed through reference-counting, which depends on the `_AddRef` and `_Release` methods inherited from `Interface`. These methods, and reference counting in general, are not applicable on the .NET platform, which is a garbage collected environment. Using the default implementation of reference counting, when an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope. Some classes implement interfaces to bypass this default lifetime management, and some hybrid objects use reference counting only when the object does not have an owner.

Global interface-type variables can be initialized only to `nil`.

To determine whether an interface-type expression references an object, pass it to the standard function `Assigned`.

## Interface Assignment Compatibility

---

Variables of a given class type are assignment-compatible with any interface type implemented by the class. Variables of an interface type are assignment-compatible with any ancestor interface type. The value `nil` can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type `IDispatch` or a descendant, the variant receives the type code `varDispatch`. Otherwise, the variant receives the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be assigned to an `IInterface` variable. A variant whose type code is `varEmpty` or `varDispatch` can be assigned to an `IDispatch` variable.

## Interface Typecasts

---

An interface-type expression can be cast to `Variant`. If the interface is of type `IDispatch` or a descendant, the resulting variant has the type code `varDispatch`. Otherwise, the resulting variant has the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be cast to `IInterface`. A variant whose type code is `varEmpty` or `varDispatch` can be cast to `IDispatch`.

## Interface Querying

---

You can use the `as` operator to perform checked interface typecasts. This is known as interface querying, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

```
object as interface
```

where `object` is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and `interface` is any interface declared with a GUID.

An interface query returns `nil` if `object` is `nil`. Otherwise, it passes the GUID of `interface` to the `QueryInterface` method in `object`, raising an exception unless `QueryInterface` returns zero. If `QueryInterface` returns zero (indicating that `object`'s class implements `interface`), the interface query returns an interface reference to `object`.

# Automation Objects (Win32 Only)

---

An object whose class implements the IDispatch interface (declared in the [System](#) unit) is an Automation object.

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include [ComObj](#) in the uses clause of one of your units or your program or library.

## Dispatch Interface Types

---

Dispatch interface types define the methods and properties that an Automation object implements through IDispatch. Calls to methods of a dispatch interface are routed through IDispatch's Invoke method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
    ['{GUID}']
    memberList
end;
```

where ['{GUID}'] is optional and memberList consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
    IStringsDisp = dispinterface
        ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
        property ControlDefault[Index: Integer]: OleVariant dispid 0;
default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
        function _NewEnum: IUnknown; dispid -4;
end;
```

## Dispatch interface methods

---

Methods of a dispatch interface are prototypes for calls to the Invoke method of the underlying IDispatch implementation. To specify an Automation dispatch ID for a method, include the dispid directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.



A method declared in a dispatch interface cannot contain directives other than `dispid`. Parameter and result types must be automatable. In other words, they must be `Byte`, `Currency`, `Real`, `Double`, `Longint`, `Integer`, `Single`, `Smallint`, `AnsiString`, `WideString`, `TDateTime`, `Variant`, `OleVariant`, `WordBool`, or any interface type.

## Dispatch interface properties

---

Properties of a dispatch interface do not include access specifiers. They can be declared as read only or write only. To specify a dispatch ID for a property, include the `dispid` directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as default. No other directives are allowed in dispatch-interface property declarations.

## Accessing Automation Objects

---

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The `CreateOleObject` function (defined in `ComObj`) returns an `IDispatch` reference to an Automation object and is assignment-compatible with the variant `Word`.

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of `varByte` are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the `VarArrayLock` and `VarArrayUnlock` routines.

## Automation Object Method-Call Syntax

---

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both positional and named parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the `:=` symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');  
Word.FileSaveAs('test.doc', 6);  
Word.FileSaveAs('test.doc',,,'secret');  
Word.FileSaveAs('test.doc', Password := 'secret');  
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type Byte, Smallint, Integer, Single, Double, Currency, TDateTime, AnsiString, WordBool, or Variant. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

## Dual Interfaces

---

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from `IDispatch`.

All methods of a dual interface (except from those inherited from `IInterface` and `IDispatch`) must use the safecall convention, and all method parameter and result types must be automatable. (The automatable types are Byte, Currency, Real, Double, Real48, Integer, Single, Smallint, AnsiString, ShortString, TDateTime, Variant, OleVariant, and WordBool.)

# Memory Management

---

This section describes memory management issues related to programming in Delphi on Win32, and on .NET.

# Memory Management on the Win32 Platform

---

The following material describes how memory management on Win32 is handled, and briefly describes memory issues of variables.

## The Memory Manager (Win32 Only)

---

The memory manager manages all dynamic memory allocations and deallocations in an application. The `New`, `Dispose`, `GetMem`, `ReallocMem`, and `FreeMem` standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

The memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of `GlobalAlloc`, `LocalAlloc`, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the `VirtualAlloc` and `VirtualFree` functions). The memory manager reserves memory from the operating system in 1Mb sections of address space, and commits memory as required in 16K increments. It decommits and releases unused memory in 16K and 1Mb sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The memory manager maintains two status variables, `AllocMemCount` and `AllocMemSize`, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging.

The `System` unit provides two procedures, `GetMemoryManager` and `SetMemoryManager`, that allow applications to intercept low-level memory manager calls. The `System` unit also provides a function called `GetHeapStatus` that returns a record containing detailed memory-manager status information.

## Variables

---

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

On Win32, an application's stack is defined by two values: the minimum stack size and the maximum stack size. The values are controlled through the `$MINSTACKSIZE` and `$MAXSTACKSIZE` compiler

directives, and default to 16,384 (16K) and 1,048,576 (1Mb) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Win32 application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an `EStackOverflow` exception is raised. (Stack overflow checking is completely automatic. The `$S` compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

Dynamic variables created with the `GetMem` or `New` procedure are heap-allocated and persist until they are deallocated with `FreeMem` or `Dispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

# Internal Data Formats

---

The following topics describe the internal formats of Delphi data types.

## Integer Types

---

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range 128..127 (Shortint), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (Byte), the variable is stored as an unsigned byte.
- If both bounds are within the range 32768..32767 (Smallint), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (Word), the variable is stored as an unsigned word.
- If both bounds are within the range 2147483648..2147483647 (Longint), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (Longword), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (Int64).

## Character Types

---

A Char, an AnsiChar, or a subrange of a Char type is stored as an unsigned byte. A WideChar is stored as an unsigned word.

## Boolean Types

---

A Boolean type is stored as a Byte, a ByteBool is stored as a Byte, a WordBool type is stored as a Word, and a LongBool is stored as a Longint.

A Boolean can assume the values 0 (False) and 1 (True). ByteBool, WordBool, and LongBool types can assume the values 0 (False) or nonzero (True).

## Enumerated Types

---

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the `{ $Z1 }` state (the default). If an enumerated type has more than 256 values, or if the type was declared in the `{ $Z2 }` state, it is stored as an unsigned word. If an enumerated type is declared in the `{ $Z4 }` state, it is stored as an unsigned double-word.

## Real Types

---

The real types store the binary representation of a sign (+ or -), an exponent, and a significand. A real value has the form

$\pm \text{significand} * 2^{\text{exponent}}$

where the *significand* has a single bit to the left of the binary decimal point. (That is,  $0 \leq \text{significand} < 2$ .)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses. For example, for a Real48 value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

## The Real48 type

A 6-byte (48-bit) Real48 number is divided into three fields:

1398  
sf e

If  $0 < e \leq 255$ , the value *v* of the number is given by

$$v = (1)^s * 2^{(e-129)} * (1.f)$$

If  $e = 0$ , then  $v = 0$ .

The Real48 type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a Real48, while NaNs and infinities produce an overflow error if an attempt is made to store them in a Real48.

**Note:** The Real48 type is not supported on the .NET platform.

## The Single type

A 4-byte (32-bit) Single number is divided into three fields

1823  
sef

The value *v* of the number is given by

if  $0 < e < 255$ , then  $v = (1)^s * 2^{(e-127)} * (1.f)$

if  $e = 0$  and  $f \neq 0$ , then  $v = (1)^s * 2^{(126)} * (0.f)$

if  $e = 0$  and  $f = 0$ , then  $v = (1)^s * 0$

if  $e = 255$  and  $f = 0$ , then  $v = (1)^s * \text{Inf}$

if  $e = 255$  and  $f \neq 0$ , then  $v$  is a NaN

## The Double type

An 8-byte (64-bit) Double number is divided into three fields

---

11152

se f

---

The value  $v$  of the number is given by

if  $0 < e < 2047$ , then  $v = (1)^s * 2^{(e-1023)} * (1.f)$

if  $e = 0$  and  $f \neq 0$ , then  $v = (1)^s * 2^{(1022)} * (0.f)$

if  $e = 0$  and  $f = 0$ , then  $v = (1)^s * 0$

if  $e = 2047$  and  $f = 0$ , then  $v = (1)^s * \text{Inf}$

if  $e = 2047$  and  $f \neq 0$ , then  $v$  is a NaN

## The Extended type

A 10-byte (80-bit) Extended number is divided into four fields:

---

115163

se i f

---

The value  $v$  of the number is given by

if  $0 \leq e < 32767$ , then  $v = (1)^s * 2^{(e-16383)} * (i.f)$

if  $e = 32767$  and  $f = 0$ , then  $v = (1)^s * \text{Inf}$

if  $e = 32767$  and  $f \neq 0$ , then  $v$  is a NaN

## The Comp type

An 8-byte (64-bit) Comp number is stored as a signed 64-bit integer.

## The Currency type

An 8-byte (64-bit) Currency number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

## Pointer Types

---

A Pointer type is stored in 4 bytes as a 32-bit address. The pointer value nil is stored as zero.



## Short String Types

---

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (`string[255]`).

## Long String Types

---

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

### *Long string dynamic memory layout*

Offset	Contents
-8	32-bit reference-count
-4	length in bytes
0..Length - 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of -1.

## Wide String Types

---

On Win32, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide string memory block on Windows.

### *Wide string dynamic memory layout (Windows)*

Offset	Contents
-4	32-bit length indicator (in bytes)
$0..Length - 1$	character string
$Length$	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

## Set Types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to

$$(Max \text{ div } 8) (Min \text{ div } 8) + 1$$

where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is

$$(E \text{ div } 8) (Min \text{ div } 8)$$

and the bit number within that byte is

$$E \text{ mod } 8$$

where *E* denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic Integer type or if the program contains code that takes the address of the set.

## Static Array Types

A static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

## Dynamic Array Types

A dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is nil and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a

dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

#### ***Dynamic array memory layout***

Offset	Contents
-8	32-bit reference-count
-4	32-bit length indicator (number of elements)
<i>0..Length</i> * (size of element) -1 array elements	

## **Record Types**

When a record type is declared in the `{ $A+ }` state (the default), and when the declaration does not include a packed modifier, the type is an unpacked record type, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field and by whether fields are declared together. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

#### ***Type alignment masks***

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for <i>Real48</i> , 4 for <i>Single</i> , 8 for <i>Double</i> and <i>Extended</i>
Short string types <sup>1</sup>	
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	determined by the <code>\$A</code> directive.

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

If two fields share a common type specification, they are packed even if the declaration does not include the packed modifier and the record type is not declared in the `{ $A- }` state. Thus, for example, given the following declaration

```

type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;

```

**A** and **B** are packed (aligned on byte boundaries) because they share the same type specification. The compiler pads the structure with unused bytes to ensure that **C** appears on a quadword boundary.

When a record type is declared in the `{ $A- }` state, or when the declaration includes the packed modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

## File Types

---

File types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```

type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of Char; );
    end;

```

Text files occupy 460 bytes, which are laid out as follows:

```

type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;

```

```

Mode: word;
Flags: word;
BufSize: Cardinal;
BufPos: Cardinal;
BufEnd: Cardinal;
BufPtr: PChar;
OpenFunc: Pointer;
InOutFunc: Pointer;
FlushFunc: Pointer;
CloseFunc: Pointer;
UserData: array[1..32] of Byte;
Name: array[0..259] of Char;
Buffer: TTextBuf;

end;

```

Handle contains the file's handle (when the file is open).

The *Mode* field can assume one of the values

```

const
  fmClosed = $D7B0;
  fmInput = $D7B1;
  fmOutput = $D7B2;
  fmInOut = $D7B3;

```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see Device functions. *Flags* determines the line break style as follows:

---

bit 0 clear LF line breaks

bit 0 set CRLF line breaks

---

All other *Flags* bits are reserved for future use.

## Procedural Types

A procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

## Class Types

A class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object's fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMT's are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMT's, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointersone per user-defined virtual method in the class typein order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Delphi's implementation. Applications should use the methods defined in TObject to query this information, since the layout is likely to change in future implementations of the Delphi language.

### Virtual method table layout

Offset	Type	Description
-76	Pointer	pointer to virtual method table (or nil)
-72	Pointer	pointer to interface table (or nil)
-68	Pointer	pointer to Automation information table (or nil)
-64	Pointer	pointer to instance initialization table (or nil)
-60	Pointer	pointer to type information table (or nil)
-56	Pointer	pointer to field definition table (or nil)
-52	Pointer	pointer to method definition table (or nil)
-48	Pointer	pointer to dynamic method table (or nil)
-44	Pointer	pointer to short string containing class name
-40	Cardinal	instance size in bytes
-36	Pointer	pointer to a pointer to ancestor class (or nil)
-32	Pointer	pointer to entry point of <i>SafecallException</i> method (or nil)

-28	Pointer	entry point of <i>AfterConstruction</i> method
-24	Pointer	entry point of <i>BeforeDestruction</i> method
-20	Pointer	entry point of <i>Dispatch</i> method
-16	Pointer	entry point of <i>DefaultHandler</i> method
-12	Pointer	entry point of <i>NewInstance</i> method
-8	Pointer	entry point of <i>FreeInstance</i> method
-4	Pointer	entry point of <i>Destroy</i> destructor
0	Pointer	entry point of first user-defined virtual method
4	Pointer	entry point of second user-defined virtual method

## Class Reference Types

A class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

## Variant Types

A variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The [System](#) and [Variants](#) units define constants and types for variants.

The *TVarData* type represents the internal structure of a Variant variable (on Windows, this is identical to the Variant type used by COM and the Win32 API). The *TVarData* type can be used in typecasts of Variant variables to access the internal structure of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the *varTypeMask* constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The *Reserved1*, *Reserved2*, and *Reserved3* fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and *varArray* bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Win32, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

# Memory Management Issues on the .NET Platform

---

The .NET Common Language Runtime is a garbage-collected environment. This means the programmer is freed (for the most part) from worrying about memory allocation and deallocation. Broadly speaking, after you allocate memory, the CLR determines when it is safe to free that memory. "Safe to free" means that no more references to that memory exist.

This topic covers the following memory management issues:

- Creating and destroying objects
- Unit initialization and finalization sections
- Unit initialization and finalization in assemblies and packages

## Constructors

---

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

**Note:** A constructor can initialize fields from its own class, prior to calling the inherited constructor.

## Finalization

---

Every class in the .NET Framework (including VCL.NET classes) inherits a method called `Finalize`. The garbage collector calls the `Finalize` method when the memory for the object is about to be freed. Since the method is called by the garbage collector, you have no control over when it is called. The asynchronous nature of finalization is a problem for objects that open resources such as file handles and database connections, because the `Finalize` method might not be called for some time, leaving these connections open.

To add a finalizer to a class, override the `strict protected Finalize` procedure that is inherited from `TObject`. The .NET platform places limits on what you can do in a finalizer, because it is called when the garbage collector is cleaning up objects. The finalizer may execute in a different thread than the thread the object was created in. A finalizer cannot allocate new memory, and cannot make calls outside of itself. If your class has references to other objects, a finalizer can refer to them (that is, their memory is guaranteed not to have been freed yet), but be aware that their state is undefined, as you do not know whether they have been finalized yet.

When a class has a finalizer, the CLR must add newly instantiated objects of the class to the finalization list. Further, objects with finalizers tend to persist in memory longer, as they are not freed when the garbage collector first determines that they are no longer actively referenced. If the object has references to other objects, those objects are also not freed right away (even if they don't have finalizers).



themselves), but must also persist in memory until the original object is finalized. Therefore, finalizers do impart a fair amount of overhead in terms of memory consumption and execution performance, so they should be used judiciously.

It is a good practice to restrict finalizers to small objects that represent unmanaged resources. Classes that use these resources can then hold a reference to the small object with the finalizer. In this way, big classes, and classes that reference many other classes, do not hoard memory because of a finalizer.

Another good practice is to suppress finalizers when a particular resource has already been released in a destructor. After freeing the resources, you can call `SuppressFinalize`, which causes the CLR to remove the object from the finalization list. Be careful not to call `SuppressFinalize` with a nil reference, as that causes a runtime exception.

## The Dispose Pattern

---

Another way to free up resources is to implement the *dispose* pattern. Classes adhering to the dispose pattern must implement the .NET interface called `IDisposable`. `IDisposable` contains only one method, called `Dispose`. Unlike the `Finalize` method, the `Dispose` method is public. It can be called directly by a user of the class, as opposed to relying on the garbage collector to call it. This gives you back control of freeing resources, but calling `Dispose` still does not reclaim memory for the object itself - that is still for the garbage collector to do. Note that some classes in the .NET Framework implement both `Dispose`, and another method such as `Close`. Typically the `Close` method simply calls `Dispose`, but the extra method is provided because it seems more "natural" for certain classes such as files.

Delphi for .NET classes are free to use the `Finalize` method for freeing system resources, however the recommended method is to implement the dispose pattern. The Delphi for .NET compiler recognizes a very specific destructor pattern in your class, and implements the `IDisposable` interface for you. This enables you to continue writing new code for the .NET platform the same way you always have, while allowing much of your existing Win32 Delphi code to run in the garbage collected environment of the CLR.

The compiler recognizes the following specific pattern of a Delphi destructor:

```
TMyClass = class(TObject)
    destructor Destroy; override;
end;
```

Your destructor must fit this pattern exactly:

- The name of the destructor must be `Destroy`.
- The keyword `override` must be specified.
- The destructor cannot take any parameters.

In the compiler's implementation of the dispose pattern, the `Free` method is written so that if the class implements the `IDisposable` interface (which it does), then the `Dispose` method is called, which in turn calls your destructor.

You can still implement the `IDisposable` interface directly, if you choose. However, the compiler's automatic implementation of the `Free-Dispose-Destroy` mechanism cannot coexist with your implementation of `IDisposable`. The two methods of implementing `IDisposable` are mutually exclusive. You must choose to either implement `IDisposable` directly, or equip your class with the familiar `destructor Destroy; override` pattern and rely on the compiler to do the rest. The `Free` method will call `Dispose` in either case, but if you implement `Dispose` yourself, you must call your destructor yourself. If you want to implement `IDisposable` yourself, your destructor cannot be called `Destroy`.

**Note:** You can declare destructors with other names; the compiler only provides the `IDisposable` implementation when the destructor fits the above pattern.

The `Dispose` method is not called automatically; the `Free` method must be called in order for `Dispose` to be called. If an object is freed by the garbage collector because there are no references to it, but you did not explicitly call `Free` on the object, the object will be freed, but the destructor will not execute.

**Note:** When the garbage collector frees the memory used by an object, it also reclaims the memory used by all fields of the object instance as well. This means the most common reason for implementing destructors in Delphi for Win32 - to release allocated memory - no longer applies. However, in most cases, unmanaged resources such as window handles or file handles still need to be released.

To eliminate the possibility of destructors being called more than once, the Delphi for .NET compiler introduces a field called `DisposeCount` into every class declaration. If the class already has a field by this name, the name collision will cause the compiler to produce a syntax error in the destructor.

## Unit Initialization and Finalization

---

On the .NET platform, units that you depend on will be initialized prior to initializing your own unit. However, there is no way to guarantee the order in which units are initialized. Nor is there a way to guarantee when they will be initialized. Be aware of initialization code that depends on another unit's initialization side effects, such as the creation of a file. Such a dependency cannot be made to work reliably on the .NET platform.

Unit finalization is subject to the same constraints and difficulties as the `Finalize` method of objects. Specifically, unit finalization is asynchronous, and, there no way to determine when it will happen (or if it will happen, though under most circumstances, it will).

Typical tasks performed in a unit finalization include freeing global objects, unregistering objects that are used by other units, and freeing resources. Because .NET is a memory managed environment, the garbage collector will free global objects even if the unit finalization section is not called. The units in an application domain are loaded and unloaded together, so you do not need to worry about unregistering objects. All units that can possibly refer to each other (even in different assemblies) are released at the same time. Since object references do not cross application domains, there is no danger of something keeping a dangling reference to an object type or code that has been unloaded from memory.

Freeing resources (such as file handles or window handles) is the most important consideration in unit finalization. Because unit finalization sections are not guaranteed to be called, you may want to rewrite your code to handle this issue using finalizers rather than relying on the unit finalization.

The main points to keep in mind for unit initialization and finalization on the .NET platform are:

1. The `Finalize` method is called asynchronously (both for objects, and for units).
2. Finalization and destructors are used to free unmanaged resources such as file handles. You do not need to destroy object member variables; the garbage collector takes care of this for you.
3. Classes should rely on the compiler's implementation of `IDisposable`, and provide a destructor called `Destroy`.
4. If a class implements `IDisposable` itself, it cannot have a destructor called `Destroy`.
5. Reference counting is deprecated. Try to use the `destructor Destroy; override;` pattern wherever possible.
6. Unit initialization should not depend on side effects produced by initialization of dependent units.

## Unit Initialization Considerations for Assemblies and Dynamically Linked Packages

---

Under Win32, the Delphi compiler uses the `DllMain` function as a hook from which to execute unit initialization code. No such execution path exists in the .NET environment. Fortunately, other means of implementing unit initialization exist in the .NET Framework. However, the differences in the implementation between Win32 and .NET could impact the order of unit initialization in your application.

The Delphi for .NET compiler uses CLS-compliant class constructors to implement unit initialization hooks. The CLR requires that every object type have a class constructor. These constructors, or type initializers, are guaranteed to be executed *at most* one time. Class constructors are executed at most one time, because in order for the type to be loaded, it must be used. That is, the assembly containing a type will not be loaded until the type is actually used at runtime. If the assembly is never loaded, its unit initialization section will never run.

Circular unit references also impact the unit initialization process. If unit A uses unit B, and unit B then uses unit A in its implementation section, the order of unit initialization is undefined. To fully understand the possibilities, it is helpful to look at the process one step at a time.

1. Unit A's initialization section uses a type from unit B. If this is the first reference to the type, the CLR will load its assembly, triggering the unit initialization of unit B.
2. As a consequence, loading and initializing unit B occurs before unit A's initialization section has completed execution. Note this is a change from how unit initialization works under Win32.
3. Suppose that unit B's initialization is in progress, and that a type from unit A is used. Unit A has not completed initialization, and such a reference could cause an access violation.

The unit initialization should only use types defined within that unit. Using types from outside the unit will impact unit initialization, and could cause an access violation, as noted above.

Unit initialization for DLLs happens automatically; it is triggered when a type within the DLL is referenced. Applications created with other .NET languages can use Delphi for .NET assemblies without concern for the details of unit initialization.

# Program Control

---

This section describes how parameters are passed to procedures and functions.

# Program Control

---

The concepts of passing parameters and function result processing are important to understand before you undertake your application projects. Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

The following topics are covered in this material:

- Passing Parameters.
- Handling Function Results.
- Handling Method Calls.
- Understanding Exit Procedures.

## Passing Parameters

---

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see the topic on Calling Conventions.

## By Value vs. By Reference

Variable (var) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (const) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A Single parameter occupies 4 bytes, and a Double, Comp, or Currency parameter occupies 8 bytes. A Real48 occupies 8 bytes, with the Real48 value stored in the lower 6 bytes. An Extended occupies 12 bytes, with the Extended value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.
- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value `nil` is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the register and pascal conventions, a variant parameter is passed as a 32-bit pointer to a Variant value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to

this rule is that records are always passed directly on the stack under the cdecl, stdcall, and safecall conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.

- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

## Pascal, cdecl, stdcall, and safecall Conventions

Under the pascal, cdecl, stdcall and safecall conventions, all parameters are passed on the stack. Under the pascal convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. Under the cdecl, stdcall, and safecall conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

## Register Convention

Under the register convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the pascal convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, Int64, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to Test passes A in EAX as a 32-bit integer, B in EDX as a pointer to a Char, and D in ECX as a pointer to a long-string memory block; C and E are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

## Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

**Note:** Delphi language procedures and functions are generally invoked with the assumption that the FPU stack is empty: The compiler tries to use all eight FPU stack entries when it generates code.

When working with the MMX and XMM instructions, be sure to preserve the values of the xmm and mm registers. Delphi functions are invoked with the assumption that the x87 FPU data registers are available for use by x87 floating point instructions. That is, the compiler assumes that the EMMS/FEMMS instruction has been called after MMX operations. Delphi functions do not make any assumptions about the state and content of xmm registers. They do not guarantee that the content of xmm registers is unchanged.

## Handling Function Results

---

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type Currency, the value in ST(0) is scaled by 10000. For example, the Currency value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, or variant result, the effects are the same as if the function result were declared as an additional var parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Int64 is returned in EDX:EAX.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional var parameter that is passed to the function after the declared parameters.

## Handling Method Calls

---

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter Self, which is a reference to the instance or class in which the method is called. The Self parameter is passed as a 32-bit pointer.

- Under the register convention, Self behaves as if it were declared before all other parameters. It is therefore always passed in the EAX register.
- Under the pascal convention, Self behaves as if it were declared after all other parameters (including the additional var parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.



- Under the cdecl, stdcall, and safecall conventions, Self behaves as if it were declared before all other parameters, but after the additional var parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional var parameter.

Constructors and destructors use the same calling conventions as other methods, except that an additional Boolean flag parameter is passed to indicate the context of the constructor or destructor call.

A value of False in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the inherited keyword. In this case, the constructor behaves like an ordinary method. A value of True in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by Self, and returns a reference to the newly created object in EAX.

A value of False in the flag parameter of a destructor call indicates that the destructor was invoked using the inherited keyword. In this case, the destructor behaves like an ordinary method. A value of True in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by Self just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the register convention, it is passed in the DL register. Under the pascal convention, it is pushed before all other parameters. Under the cdecl, stdcall, and safecall conventions, it is pushed just before the Self parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that BeforeDestruction or AfterConstruction can be called properly.

## Understanding Exit Procedures

---

Exit procedures ensure that specific actions such as updating and closing files are carried out before a program terminates. The `ExitProc` pointer variable allows you to *install* an exit procedure, so that it is always called as part of the program's termination whether the termination is normal, forced by a call to `Halt`, or the result of a runtime error. An exit procedure takes no parameters.

**Note:** It is recommended that you use finalization sections rather than exit procedures for all exit behavior. `Exit` procedures are available only for executables. For .DLLs (Win32) you can use a similar variable, `DllProc`, which is called when the library is loaded as well as when it is unloaded. For packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents

of `ExitProc` before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of `ExitProc`.

The following code shows a skeleton implementation of an exit procedure.

```
var
  ExitSave: Pointer;

procedure MyExit;

begin
  ExitProc := ExitSave; // always restore old vector first
  .
  .
  .
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  .
  .
  .
end.
```

On entry, the code saves the contents of `ExitProc` in `ExitSave`, then installs the `MyExit` procedure. When called as part of the termination process, the first thing `MyExit` does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until `ExitProc` becomes nilnil. To avoid infinite loops, `ExitProc` is set to nil before every call, so the next exit procedure is called only if the current exit procedure assigns an address to `ExitProc`. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the `ExitCode` integer variable and the `ErrorAddr` pointer variable. In case of normal termination, `ExitCode` is zero and `ErrorAddr` is nil. In case of termination through a call to `Halt`, `ExitCode` contains the value passed to `Halt` and `ErrorAddr` is nil. In case of termination due to a runtime error, `ExitCode` contains the error code and `ErrorAddr` contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the Input and Output files. If `ErrorAddr` is not nil, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines `ErrorAddr` and outputs a message if it's not nil; before returning, set `ErrorAddr` to nil so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in `ExitCode` as a return code.

## Inline Assembly Code (Win32 Only)

---

This section describes the use of the inline assembler on the Win32 platform.

# Using Inline Assembly Code (Win32 Only)

---

The built-in assembler allows you to write assembly code within Delphi programs. The inline assembler is available only on the Win32 Delphi compiler. It has the following features:

- Allows for inline assembly.
- Supports all instructions found in the Intel Pentium III, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!).
- Provides no macro support, but allows for pure assembly function procedures.
- Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements.

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See the topic on External declarations for more information. If you have external assembly code that you want to use in your applications, you should consider rewriting it in the Delphi language or minimally reimplement it using the inline assembler.

## Using the asm Statement

---

The built-in assembler is accessed through asm statements, which have the form

```
asm statementList end
```

where *statementList* is a sequence of assembly statements separated by semicolons, end-of-line characters, or Delphi comments.

Comments in an asm statement must be in Delphi style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word `inline` and the directive `assembler` are maintained for backward compatibility only. They have no effect on the compiler.

## Using Registers

---

In general, the rules of register use in an asm statement are the same as those of an external procedure or function. An asm statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an asm statement, EBP points to the current stack frame and ESP points to the top of the stack. Except for ESP and EBP, an asm statement can assume nothing about register contents on entry to the statement.

# Understanding Assembler Syntax (Win32 Only)

---

The inline assembler is available only on the Win32 Delphi compiler. The following material describes the elements of the assembler syntax necessary for proper use.

- Assembler Statement Syntax
- Labels
- Instruction Opcodes
- Assembly Directives
- Operands

## Assembler Statement Syntax

---

This syntax of an assembly statement is

```
Label: Prefix Opcode Operand1, Operand2
```

where *Label* is a label, *Prefix* is an assembly prefix opcode (operation code), *Opcode* is an assembly instruction opcode or directive, and *Operand* is an assembly expression. Label and Prefix are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembly statements, but not within them. For example,

```
        MOV AX,1 {Initial value}           { OK }
        MOV CX,100 {Count}
{ OK }

        MOV {Initial value} AX,1;
{ Error! }
        MOV CX, {Count} 100
{ Error! }
```

## Labels

---

Labels are used in built-in assembly statements as they are in the Delphi language by writing the label and a colon before a statement. There is no limit to a label's length. As in Delphi, labels must be declared in a label declaration part in the block containing the asm statement. The one exception to this rule is local labels.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to asm statements, and the scope of a local label extends from the asm reserved word to the end of the asm statement that contains it. A local label doesn't have to be declared.

## Instruction Opcodes

---

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

## RET instruction sizing

The RET instruction opcode always generates a near return.

## Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is 128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is 128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembly statement

```
JC      Stop
```

where Stop isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
JNC     Skip
JMP     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

## Assembly Directives

---

The built-in assembler supports three assembly define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between 128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between 32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between 2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The DQ directive defines a quad word for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembly statements. To generate uninitialized or initialized data in the data segment, you should use Delphi var or const declarations.

Some examples of DB, DW, and DD directives follow.

```
asm
    DB
FFH
                                { One byte }
    DB
0,99
                                { Two bytes }
    DB
'A'
                                { Ord('A') }
    DB      'Hello world...',0DH,0AH      { String followed by CR/LF }
    DB      12,
'string'
style string }
    DW
0FFFFH
{ One word }
```

```

        DW
0,9999          { Two words }

        DW
'A'            { Same as DB 'A',0 }

        DW
'BA'          { Same as DB 'A','B' }

        DW
MyVar          { Offset of MyVar }

        DW
MyProc
{ Offset of MyProc }
        DD
0FFFFFFFFH
{ One double-word }
        DD
0,9999999999
{ Two double-words }
        DD
'A'            { Same as DB 'A',0,0,0 }

        DD
'DCBA'
{ Same as DB 'A','B','C','D' }
        DD
MyVar          { Pointer to MyVar }

        DD
MyProc
{ Pointer to MyProc }
end;

```

When an identifier precedes a DB, DW , or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, the assembler allows the following:

```

ByteVar      DB      ?
WordVar      DW      ?
IntVar       DD      ?

.
.
.
MOV          AL,ByteVar
MOV          BX,WordVar
MOV          ECX,IntVar

```



The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembly statement is a label. All variables must be declared using Delphi syntax; the preceding construction can be replaced by

```
var
    ByteVar: Byte;
    WordVar: Word;
    IntVar: Integer;
    .
    .
    .

asm
    MOV AL,ByteVar
    MOV BX,WordVar
    MOV ECX,IntVar
end;
```

SMALL and LARGE can be used determine the width of a displacement:

```
MOV EAX, [LARGE $1234]
```

This instruction generates a 'normal' move with a 32-bit displacement (\$00001234).

```
MOV EAX, [SMALL $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement (\$1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
MOV EAX, [SMALL 123]
```

as opposed to

```
MOV EAX, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual methods: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter (for example, TExample.VirtualMethod), or an interface name and an interface method name.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.

DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

**Note:** Methods with the *message* directive are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
TMyClass = class
  procedure x; message MYMESSAGE;
end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;
type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

  procedure TExample.DynamicMethod;
begin
  end;

  procedure TExample.VirtualMethod;
begin
  end;

  procedure CallDynamicMethod(e: TExample);
asm
  // Save ESI register
  PUSH    ESI

  // Instance pointer needs to be in EAX
  MOV     EAX, e

  // DMT entry index needs to be in (E)SI
  MOV     ESI, DMTINDEX TExample.DynamicMethod

  // Now call the method
  CALL    System.@CallDynaInst

  // Restore ESI register
  POP     ESI

end;
```

```

                                procedure CallVirtualMethod(e: TExample);
asm
    // Instance pointer needs to be in EAX
    MOV     EAX, e

                                // Retrieve VMT table entry
    MOV     EDX, [EAX]

                                // Now call the method at offset VMTOFFSET
    CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]

                                end;

                                var
    e: TExample;
begin
    e := TExample.Create;
    try
        CallDynamicMethod(e);
        CallVirtualMethod(e);
    finally
        e.Free;
    end;
end.

```

## Operands

Inline assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings:

### ***Built-in assembler reserved words***

AH	CL	DX	ESP	mm4	SHL	WORD
AL	CS	EAXFS		mm5	SHR	xmm0
AND	CX	EBPGS		mm6	SI	xmm1
AX	DH	EBXHIGH		mm7	SMALL	xmm2
BH	DI	ECXLARGEMOD			SP	xmm3
BL	DL	EDI LOW	NOT	SS		xmm4
BP	CL	EDXmm0	OFFSETST			xmm5
BX	DMTINDEXEIP	mm1	OR	TBYTE		xmm6
BYTEDS		ES	mm2	PTR	TYPE	xmm7
CH	DWORD	ESI	mm3	QWORD	VMTOFFSETXOR	

Reserved words always take precedence over user-defined identifiers. For example,

```
var
    Ch: Char;
    .
    .
    .
asm
    MOV    CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) override operator:

```
MOV&Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

# Assembly Expressions (Win32 Only)

---

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants. The inline assembler is available only on the Win32 Delphi compiler.

Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type. This topic covers the following material:

- Differences between Delphi and Assembler Expressions
- Expression Elements
- Expression Classes
- Expression Types
- Expression Operators

## Differences between Delphi and Assembler Expressions

---

The most important difference between Delphi expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value. In other words, it must resolve to a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid statement.

```
asm
  MOV     Z, X+Y
end;
```

Because both `X` and `Y` are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves the value 30 into the variable `Z`. But if `X` and `Y` are variables

```
var
  X, Y: Integer;
```

the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of `X` and `Y` into `Z` you would use

```
asm
  MOV      EAX, X
  ADD      EAX, Y
  MOV      Z, EAX
end;
```

In a Delphi expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Delphi the expression `X + 4` (where `X` is a variable) means the contents of `X` plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of `X`. So, even though you are allowed to write

```
asm
  MOV      EAX, X+4
end;
```

this code doesn't load the value of `X` plus 4 into `AX`; instead, it loads the value of a word stored four bytes beyond `X`. The correct way to add 4 to the contents of `X` is

```
asm
  MOV      EAX, X
  ADD      EAX, 4
end;
```

## Expression Elements

---

The elements of an expression are constants, registers, and symbols.

### Numeric Constants

Numeric constants must be integers, and their values must be between 2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a `B` after the number, octal notation by writing an `O` after the number, and hexadecimal notation by writing an `H` after the number or a `$` before the number.

Numeric constants must start with one of the digits 0 through 9 or the `$` character. When you write a hexadecimal constant using the `H` suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, `0BAD4H` and `$BAD4` are hexadecimal constants, but `BAD4H` is an identifier because it starts with a letter.

## String Constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
'Linux'
"That's all folks"
'"That"'s all folks," he said.'
'100'
'''
'''
```

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

$$\text{Ord}(\text{Ch1}) + \text{Ord}(\text{Ch2}) \text{ shl } 8 + \text{Ord}(\text{Ch3}) \text{ shl } 16 + \text{Ord}(\text{Ch4}) \text{ shl } 24$$

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

### *String examples and their values*

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

## Registers

The following reserved symbols denote CPU registers in the inline assembler:

## CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers for example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(X), where X is a constant between 0 and 7 indicating the distance from the top of the register stack.

## Symbols

The built-in assembler allows you to access almost all Delphi identifiers in assembly language expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol @Result, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
    Result := X + Y;
end;
```

could be written in assembly language as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
    asm
        MOV     EAX, X
        ADD     EAX, Y
        MOV     @Result, EAX
    end;
end;
```



The following symbols cannot be used in asm statements:

- Standard procedures and functions (for example, WriteLn and Chr).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The @Result symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in asm statements.

***Symbols recognized by the built-in assembler***

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable or address of a pointer to the variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV      EAX, Count
```

assembles into `MOV EAX, [EBP4]`.

The built-in assembler treats var parameters as a 32-bit pointers, and the size of a var parameter is always 4. The syntax for accessing a var parameter is different from that for accessing a value parameter. To access the contents of a var parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
    asm
        MOV      EAX, X
        MOV      EAX, [EAX]
```

```

                                MOV          EDX, Y
                                ADD          EAX, [EDX]
                                MOV          @Result, EAX
    end;
end;

```

Identifiers can be qualified within asm statements. For example, given the declarations

```

type
    TPoint = record
        X, Y: Integer;
    end;
    TRect = record
        A, B: TPoint;
    end;
var
    P: TPoint;
    R: TRect;

```

the following constructions can be used in an asm statement to access fields.

```

MOV          EAX, P.X
MOV          EDX, P.Y
MOV          ECX, R.A.X
MOV          EBX, R.B.Y

```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```

MOV          EAX, (TRect PTR [EDX]).B.X
MOV          EAX, TRect ([EDX]).B.X
MOV          EAX, TRect [EDX].B.X
MOV          EAX, [EDX].TRect.B.X

```

## Expression Classes

---

The built-in assembler divides expressions into three classes: registers, memory references, and immediate values.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Delphi's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Delphi's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  .
  .
  .
asm
  MOV      EAX, Start                      { MOV EAX, xxxx }
  MOV      EBX, Count                     { MOV EBX, [xxxx] }
  MOV      ECX, [Start]                   { MOV ECX, [xxxx] }
  MOV      EDX, OFFSET Count               { MOV EDX, xxxx }
end;
```

Because `Start` is an immediate value, the first `MOV` is assembled into a move immediate instruction. The second `MOV`, however, is translated into a move memory instruction, as `Count` is a memory reference. In the third `MOV`, the brackets convert `Start` into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth `MOV`, the `OFFSET` operator converts `Count` into an immediate value (the offset of `Count` in the data segment).

The brackets and `OFFSET` operator complement each other. The following `asm` statement produces identical machine code to the first two lines of the previous `asm` statement.

```
asm
  MOV      EAX, OFFSET [Start]
  MOV      EBX, [OFFSET Count]
end;
```

Memory references and immediate values are further classified as either relocatable or absolute. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

## Expression Types

---

Every built-in assembler expression has a type or, more correctly, a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an Integer variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
.
.
.
asm
  MOV     AL, QuitFlag
  MOV     BX, OutBufPtr
end;
```

the assembler checks that the size of `QuitFlag` is one (a byte), and that the size of `OutBufPtr` is two (a word). The instruction

```
MOV     DL, OutBufPtr
```

produces an error because DL is a byte-sized register and `OutBufPtr` is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV     DL, BYTE PTR OutBufPtr
MOV     DL, Byte (OutBufPtr)
MOV     DL, OutBufPtr.Byte
```

These MOV instructions all refer to the first (least significant) byte of the `OutBufPtr` variable.

In some cases, a memory reference is untyped. One example is an immediate value (`Buffer`) enclosed in square brackets:

```
procedure Example(var Buffer);
asm
  MOV AL,    [Buffer]
  MOV CX,    [Buffer]
  MOV EDX,   [Buffer]
end;
```

The built-in assembler permits these instructions, because the expression `[Buffer]` has no type it just means "the contents of the location indicated by `Buffer`," and the type can be determined from the first operand (byte for AL, word for CX, and double-word for EDX).

In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast. For example,

```
INC      BYTE PTR [ECX]
IMUL     WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Delphi types.

#### ***Predefined type symbols***

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

## **Expression Operators**

The built-in assembler provides a variety of operators. Precedence rules are different from that of the Delphi language; for example, in an asm statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

#### ***Precedence of built-in assembler expression operators***

Operators	Remarks	Precedence
&		highest
(... ), [... ],,, HIGH, LOW		
+, -	unary + and -	
:		
OFFSET, TYPE, PTR, *, /, MOD, SHL, SHR, +, -	binary + and -	
NOT, AND, OR, XOR		lowest

The following table defines the built-in assembler's expression operators.

#### ***Definitions of built-in assembler expression operators***

Operator	Description
&	<b>Identifier override.</b> The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(... )	<b>Subexpression.</b> Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[... ]	<b>Memory reference.</b> The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	<b>Structure member selector.</b> The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	<b>Unary plus.</b> Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	<b>Unary minus.</b> Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	<b>Addition.</b> The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
-	<b>Subtraction.</b> The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	<b>Segment override.</b> Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	<b>Typecast operator.</b> The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	<b>Multiplication.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	<b>Integer division.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.

MOD	<b>Remainder after integer division.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	<b>Logical shift left.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	<b>Logical shift right.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	<b>Bitwise negation.</b> The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	<b>Bitwise AND.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	<b>Bitwise OR.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	<b>Bitwise exclusive OR.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.

# Assembly Procedures and Functions (Win32 Only)

---

You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement. This topic covers these issues:

- Compiler Optimizations.
- Function Results.

The inline assembler is available only on the Win32 Delphi compiler.

## Compiler Optimizations

---

An example of the type of function you can write is as follows:

```
function LongMul(X, Y: Integer): Longint;
asm
    MOV     EAX, X
    IMUL   Y
end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were var parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the `@Result` symbol is an error. For strings, variants, and interfaces, the caller always allocates an `@Result` pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- Locals is the size of the local variables and Params is the size of the parameters. If both Locals and Params are zero, there is no entry code, and the exit code consists simply of a RET instruction.

The automatically generated entry and exit code for the routine looks like this:

```
PUSH     EBP                                ;Present if Locals
<> 0 or Params <> 0
MOV      EBP,ESP                            ;Present if Locals <> 0 or
Params <> 0
SUB      ESP,Locals                        ;Present if Locals <> 0
.
.
.
MOV      ESP,EBP                            ;Present if Locals <> 0
POP      EBP                                ;Present if Locals
```



```
<> 0 or Params <> 0  
RET          Params          ;Always present
```

If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

## Function Results

---

Assembly language functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (Currency values are scaled by 10000.)
- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by `@Result`.

## **.NET Topics**

---

This section contains information specific to programming in Delphi on the .NET platform.

# Using .NET Custom Attributes

---

.NET framework assemblies are self-describing entities. They contain intermediate code that is compiled to native machine instructions when the assembly is loaded. More than that, assemblies contain a wealth of information about that code. The compiler emits this descriptive information, or metadata, into the assembly as it processes the source code. In other programming environments, there is no way to access metadata once your code is compiled; the information is lost during the compilation process. On the .NET platform, however, you have the ability to access metadata using runtime reflection services.

The .NET framework gives you the ability to extend the metadata emitted by the compiler with your own descriptive attributes. These customized attributes are somewhat analogous to language keywords, and are stored with the other metadata in the assembly.

- Declaring custom attributes
- Using custom attributes
- Custom attributes and interfaces

## Declaring a Custom Attribute Class

---

Creating a custom attribute is the same as declaring a class. The custom attribute class has a constructor, and properties to set and retrieve its state data. Custom attributes must inherit from `TCustomAttribute`. The following code declares a custom attribute with a constructor and two properties:

```
type
  TCustomCodeAttribute = class(TCustomAttribute)
  private
    Fprop1 : integer;
    Fprop2 : integer;
    aVal   : integer;
    procedure Setprop1(p1 : integer);
    procedure Setprop2(p2 : integer);
  public
    constructor Create(const myVal : integer);
    property prop1 : integer read Fprop1 write Setprop1;
    property prop2 : integer read Fprop2 write Setprop2;
  end;
```

The implementation of the constructor might look like

```
constructor TCustomCodeAttribute.Create(const myVal : integer);
begin
  inherited Create;
  aVal := myVal;
end;
```

Delphi for .NET supports the creation of custom attribute classes, as shown above, and all of the custom attributes provided by the .NET framework.

## Using Custom Attributes

---

Custom attributes are placed directly before the source code symbol to which the attribute applies.

Attributes can be placed before

- variables and constants
- procedures and functions
- function results
- procedure and function parameters
- types
- fields, properties, and methods

Note that Delphi for .NET supports the use of named properties in the initialization. These can be the names of properties, or of public fields of the custom attribute class. Named properties are listed after all of the parameters required by the constructor. For example

```
[TCustomCodeAttribute(1024, prop1=512, prop2=128)]
TMyClass = class(TObject)
...
end;
```

applies the custom attribute declared above to the class `TMyClass`.

The first parameter, 1024, is the value required by the constructor. The second two parameters are the properties defined in the custom attribute.

When a custom attribute is placed before a list of multiple variable declarations, the attribute applies to all variables declared in that list. For example

```
var
  [TCustomAttribute(1024, prop1=512, prop2=128)]
  x, y, z: Integer;
```

would result in `TCustomAttribute` being applied to all three variables, `x`, `y`, and `z`.

Custom attributes applied to types can be detected at runtime with the `GetCustomAttributes` method of the `Type` class. The following Delphi code demonstrates how to query for custom attributes at runtime.

```
var
  F: TMyClass;           // TMyClass declared above
  T: System.Type;
  A: array of TObject;  // Will hold custom attributes
```

```

    I: Integer;

begin
    F := TMyClass.Create;
    T := F.GetType;
    A := T.GetCustomAttributes(True);

    // Write the type name, and then loop over custom
    // attributes returned from the call to
    // System.Type.GetCustomAttributes.
    Writeln(T.FullName);
    for I := Low(A) to High(A) do
        Writeln(A[I].GetType.FullName);
end.

```

## Using the DllImport Custom Attribute

---

You can call unmanaged Win32 APIs (and other unmanaged code) by prefixing the function declaration with the `DllImport` custom attribute. This attribute resides in the `System.Runtime.InteropServices` namespace, as shown below:

```

Program HelloWorld2;

    // Don't forget to include the InteropServices unit when using the
DllImport attribute.
    uses System.Runtime.InteropServices;

    [DllImport('user32.dll')]
    function MessageBeep(uType : LongWord) : Boolean; external;

begin
    MessageBeep(LongWord(-1));
end.

```

Note the `external` keyword is still required, to replace the block in the function declaration. All other attributes, such as the calling convention, can be passed through the `DllImport` custom attribute.

## Custom Attributes and Interfaces

---

Delphi syntax dictates that the GUID (if present) must immediately follow the declaration of an interface. Since the GUID syntax is similar to that of custom attributes, the compiler must be made to know the difference between a custom attribute - which applies to the next declaration - and a GUID specifier, which applies to the previous declaration. Without this special case, the compiler would try to apply an attribute to the first member of the interface.

When the compiler sees an interface declaration, the next square bracket construct found is assumed to be that of a GUID specifier for the interface. The GUID must be in the traditional Delphi form:

```
[ '{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}' ]
```

Alternatively, you can use the `Guid` custom attribute of the .NET framework ( `GuidAttribute`). If you choose this method, then you should introduce the attribute before the interface, as with any other custom attribute.

The effect in either case is the same: the GUID is emitted into the metadata for the interface type. Note that GUIDs are not required for interfaces in the .NET Framework. They are only used for COM interoperability.

**Note:** When importing COM interfaces with the `ComImport` custom attribute, you must declare the `GuidAttribute` instead of using the Delphi syntax.

# IDE command-line options

---

Describes available options when starting the IDE from the command line.

## IDE command-line switches

---

The following options are available when starting the IDE from the command line. You must precede all options (unless otherwise noted) with either a dash (-) or a slash (/). The options are not case-sensitive. Therefore, the following options are all identical: -d /d -D /D.

You use these options with the appropriate IDE startup command: `bds.exe`

For example:

```
bds.exe -ns -hm
```

Starts the IDE with no splash screen and tracks memory allocation.

```
bds.exe -sdc:\test\source -d c:\test\myprog.exe -td
```

Starts the IDE and loads `c:\test\myprog.exe` into the debugger and used `c:\test\source` as the location for the source code while debugging. The `-td` and any other argument that appears after the `-d` option is used as an argument to `c:\test\myprog.exe`.

## General options

---

OptionDescription	
?	Launches the IDE and displays online help for IDE command-line options.
--	Ignore rest of command-line.
hm	Heap Monitor. Displays information in the IDE title bar regarding the amount of memory allocated using the memory manager. Displays the number of blocks and bytes allocated. Information gets updated when the IDE is idle.
hv	Heap Verify. Performs validation of memory allocated using the memory manager. Displays error information in the IDE title bar if errors are found in the heap.
ns	No splash screen. Suppresses display of the splash screen during IDE startup.
np	No Project. Suppresses loading of any desktop files on IDE startup and suppresses creation of a default project.

## Debugger options

---

Option	Description
<code>attach:%1;%2</code>	Performs a debug attach, using %1 as the process ID to attach to and %2 as the event ID for that process. It can be used manually, but is used mostly for Just in Time debugging.
<code>dexename</code>	Loads the specified executable (exename) into the debugger. Any parameters specified after the exename are used as parameters to the program being debugged and are ignored by the IDE. A space is allowed between the d and the exename.
<code>rregkey</code>	Allows you specify an alternate base registry key so you can run two copies of the IDE using different configurations. This allows component developers to debug a component at design-time by using the IDE as the hosting application without the debugging IDE interfering by trying to load the component package being developed

The following options can only be used with the -d option:

<code>hhostname</code>	Hostname. Must be used with the d option. When specified, a remote debug session is initiated using the specified host name as the remote host to debug on. The remote debug server program must be running on the remote host.
<code>l</code>	(Lowercase L) Assembler startup. Do not execute startup code. Must be used with the d option. Normally, when you specify the d option, the debugger attempts to run the process to either main or WinMain. When l is specified, the process is merely loaded and no startup code is executed.
<code>sddirectories</code>	Source Directories. Must be used with the d option. The argument is either a single directory or a semicolon delimited list of directories which are used as the Debug Source Path setting (can also be set using the Project Options Directories/Conditionals option page). No space is allowed between sd and the directory list argument.
<code>td</code>	TDGoodies. Implements several features found in the Turbo Debugger, TD32 (available on Windows only). It must be used with the d option. It causes the CPU and FPU views to stay open when a process terminates. It causes Run Program Reset to terminate the current process and reload it in the debugger. If there is no current process, Run Program Reset reloads the last process that terminated. It also causes breakpoints and watches to be saved in the default desktop if desktop saving is on and no project is loaded.

## Project options

---

Option	Description
<code>filename</code>	(No preceding dash) The specified filename is loaded in the IDE. It can be a project, project group, or a single file.
<code>b</code>	AutoBuild. Must be used with the filename option. When specified, the project or project group is built automatically when the IDE starts. Any hints, errors, or warnings are then saved to a file. Then the IDE exits. This facilitates doing builds in batch mode from a batch file. The Error Level is set to 0 for successful



builds and 1 for failed builds. By default, the output file has the same name as the filename specified with the file extension changed to .err. This can be overridden using the o option.

---

m        AutoMake. Same as AutoBuild, but a make is performed rather than a full build.

---

o*outputfile* Output file. Must be used the b or m option. When specified, any hints, warnings, or errors are written to the file specified instead of the default file.

---