Chapter 1

Delphi 7 and Its IDE

In a visual programming tool such as Delphi, the role of the integrated development environment (IDE) is at times even more important than the programming language. Delphi 7 provides some interesting new features on top of the rich IDE of Delphi 6. This chapter examines these new features, as well as features added in other recent versions of Delphi. We'll also discuss a few traditional Delphi features that are not well known or obvious to newcomers. This chapter isn't a complete tutorial of the IDE, which would require far too much space; it's primarily a collection of tips and suggestions aimed at the average Delphi user.

If you *are* a beginning programmer, don't be afraid. The Delphi IDE is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi Companion Tools CD) with a tutorial that introduces the development of Delphi applications. You can find a simpler introduction to Delphi and its IDE in my *Essential Delphi* online book (discussed in Appendix C, "Free Companion Books on Delphi"). Throughout this book, I'll assume you already know how to carry out the basic hands-on operations of the IDE; all the chapters after this one focus on programming issues and techniques.

This chapter covers the following topics:

- Moving around the IDE
- The editor
- The Code Insight technology
- Designing forms
- The Project Manager
- Delphi files

Editions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The "Personal" edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi.
- The "Professional Studio" edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic web server support (WebBroker), and some of the external tools, including ModelMaker and IntraWeb. This book generally assumes you are working with at least the Professional edition.
- The "Enterprise Studio" edition is aimed at developers building enterprise applications. It includes all the XML and advanced web services technologies, CORBA support, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.
- The "Architect Studio" edition adds to the Enterprise edition support for Bold, an environment for building applications that are driven at run time by a UML model and capable of mapping their objects both to a database and to the user interface, thanks to a plethora of advanced components. Bold support is not covered in this book.

Besides the different editions available, there are ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which may be reflected in some of the screen shots.

The Professional and higher versions of Delphi 7 include a working copy of Kylix 3, in the Delphi language edition. Other than references to the CLX library and cross-platform features of Delphi, this book doesn't cover Kylix and Linux development. You can refer to *Mastering Kylix 2* (Sybex, 2002) for more information on the topic. (There aren't many differences between Kylix 2 and Kylix 3 in the Delphi language version. The most important new feature of Kylix 3 is its support of the C++ language.)

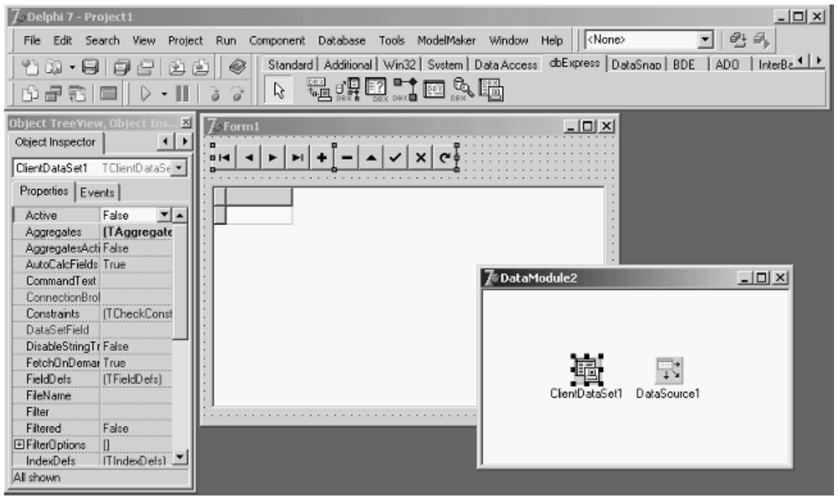
An Overview of the IDE

When you work with a visual development environment, your time is spent in two different portions of the application: visual designers and the code editor. Designers let you work with components at the visual level (such as when you place a button on a form) or at a non-visual level (such as when you place a DataSet component on a data module). You can see a form and a data module in action in Figure 1.1. In both cases, designers allow you to choose the components you need and set the initial value of the components' properties.

The code editor is where you write code. The most obvious way to write code in a visual environment involves responding to events, beginning with events attached to operations performed by program users, such as clicking on a button or selecting an item of a list box. You can use the same approach to handle internal events, such as events involving database changes or notifications from the operating system.

As programmers become more knowledgeable about Delphi they often begin by writing mainly event-handling code and then move to writing their own classes and components, and often end up spending most of their time in the editor. Because this book covers more than visual programming, and tries to help you master the entire power of Delphi, as the text proceeds you'll see more code and fewer forms.

FIGURE 1.1
A form and a data module in the Delphi 7 IDE


Delphi 7 - Project1
File Edit Search View Project Run Component Database Tools ModelMaker Window Help
<None>
Standard Additional Win32 System Data Access dbExpress DataSnap BDE ADO InterBa
Object TreeView, Object Ins...
Object Inspector
Properties Events
All shown
Form1
DataModule2
ClientDataSet1
DataSource1

ClientDataSet1 TClientDataSe

Active	False
Aggregates	(TAggregate
AggregatesActi	False
AutoCalcFields	True
CommandText	
ConnectionBrol	
Constraints	(TCheckConst
DataSetField	
DisableStringTr	False
FetchOnDemar	True
FieldDefs	(TFieldDefs)
FileName	
Filter	
Filtered	False
⊞FilterOptions	[]
IndexDefs	(TIndexDefs)

An IDE for Two Libraries

An important change appeared for the first time in Delphi 6. The IDE now lets you work on two different visual libraries: VCL (Visual Component Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File ➢ New ➢ Application command for a classic VCL-based Windows program and with the File ➢ New ➢ CLX Application command for a new CLX-based portable application.

NOTE *CLX is Delphi's Cross Platform library, which allows you to recompile your code with Kylix to run under Linux. You can read more about VCL versus CLX in Chapter 5, "Visual Controls." Using CLX is even more interesting in Delphi 7, because the Delphi language version of Kylix ships with the Windows product.*

When you create a new project or open an existing one, the Component Palette is arranged to show only the controls related to the current library (although most of the controls are shared). When you work with a non-visual designer (such as a data module), the tabs of the Component Palette that host only visual components are hidden from view.

Desktop Settings

Programmers can customize the Delphi IDE in various ways—typically, opening many windows, arranging them, and docking them to each other. However, you'll often need to open one set of windows at design time and a different set at debug time. Similarly, you might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi lets you save a given arrangement of IDE windows (called a *desktop*, or a Global Desktop, to differentiate from a Project Desktop) with a name and restore it easily. You can also make one of these groupings your default debugging setting, so that it will be restored automatically when you start the debugger. All these features are available in the Desktops toolbar. You can also work with desktop settings using the View ➢ Desktops menu.

Desktop setting information is saved in DST files (stored in Delphi's `bin` directory), which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78

[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1

[AlignmentPalette]
Create=1
Visible=0
...
```

Desktop settings override project settings, which are saved in a DSK file with a similar structure. Desktop settings help eliminate problems that can occur when you move a project between machines (or between developers) and have to rearrange the windows to your liking. Delphi separates per-user global desktop settings and per-project desktop settings, to better support team development.

TIP *If you open Delphi and cannot see the form or other windows, I suggest you try checking (or deleting) the desktop settings (from Delphi's* `bin` *directory). If you open a project received by a different user and cannot see some of the windows or dislike the desktop layout, reload your global desktop settings or delete the project DSK file.*

Environment Options

Quite a few recent updates relate to the commonly used Environment Options dialog box. The pages of this dialog box were rearranged in Delphi 6, moving the Form Designer options from the Preferences page to the new Designer page. In Delphi 6 there were also a few new options and pages:

- The Preferences page of the Environment Options dialog box has a check box that prevents Delphi windows from automatically docking with each other.

- The Environment Variables page allows you to see system environment variables (such as the standard pathnames and OS settings) and set user-defined variables. The nice point is that you can use both system- and user-defined environment variables in each of the dialog boxes of the IDE—for example, you can avoid hard-coding commonly used pathnames, replacing them with a variable. In other words, the environment variables work similarly to the `$DELPHI` variable, referring to Delphi's base directory, but can be defined by the user.
- In the Internet page you can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

About Menus

The main Delphi menu bar (which in Delphi 7 has a more modern look) is an important way to interact with the IDE, although you'll probably accomplish most tasks using shortcut keys and shortcut menus. The menu bar doesn't change much in reaction to your current operations: You need to click the right mouse button for a full list of the operations you can perform on the current window or component.

The menu bar can change considerably depending on third-party tools and wizards you've installed. In Delphi 7, ModelMaker has its own menu. You'll see other menus by installing popular add-ons like GExperts or even my own wizards (see Appendix B, "Extra Delphi Tools from other Sources" and A, "Extra Delphi Tools by the Author," respectively, for more details).

A relevant menu added to Delphi in recent editions is the Window menu in the IDE. This menu lists the open windows; previously, you could obtain this list using the Alt+0 key combination or the View ➢ Window List menu item. The Window menu is really handy, because windows often end up behind others and are hard to find. You can control the alphabetic sort order of this menu using a setting in the Windows Registry: Look for the `Main Window` subkey of Delphi (under `HKEY_CURRENT_USER\Software\Borland\Delphi\7.0`). This Registry key uses a string (in place of Boolean values), where '-1' and 'True' indicate true and '0' and 'False' indicate false.

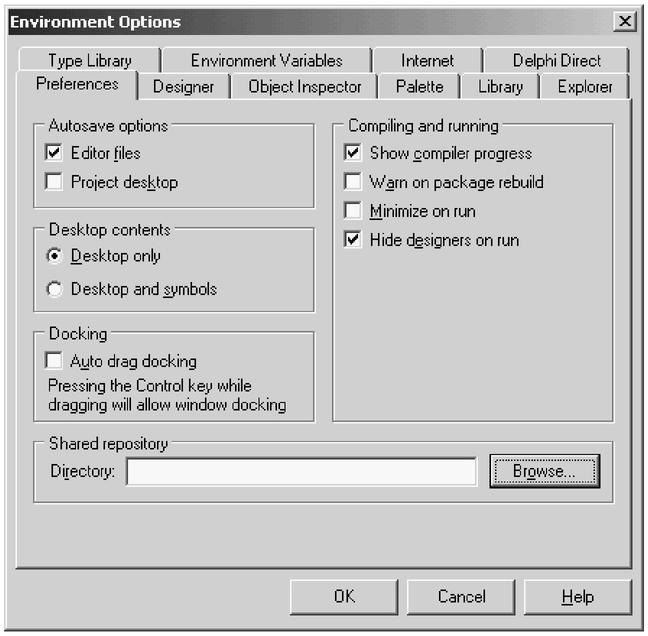
TIP *In Delphi 7, the Window menu ends with a new command: Next Window. This command is particularly useful in the form of a shortcut, Alt+End. Jumping around the various windows of the IDE has never been so simple (at least, without add-on tools).*

The Environment Options Dialog Box

As I've mentioned, some of the IDE settings require you to edit the Registry directly. I'll discuss a few more of these settings in this chapter. Of course, the most common settings can be easily tuned using the Environment Options dialog box, which is available in the Tools menu along with the Editor Options and the Debugger Options. Most of the settings are quite intuitive and well described in the Delphi Help file. Figure 1.2 shows my standard settings for the Preferences page of this dialog box.

The To-Do List

Another feature added in Delphi 5 but still quite underused is the to-do list. This is a list of tasks you still have to do to complete a project—it's a collection of notes for the programmer (or programmers; this tool can be very handy in a team). Although the idea is not new, the key concept of the to-do list in Delphi is that it works as a two-way tool.



Autosave options
- [✓] Editor files
- [ ] Project desktop

Desktop contents
- (●) Desktop only
- ( ) Desktop and symbols

Docking
- [ ] Auto drag docking

Pressing the Control key while dragging will allow window docking

Compiling and running
- [✓] Show compiler progress
- [ ] Warn on package rebuild
- [ ] Minimize on run
- [✓] Hide designers on run

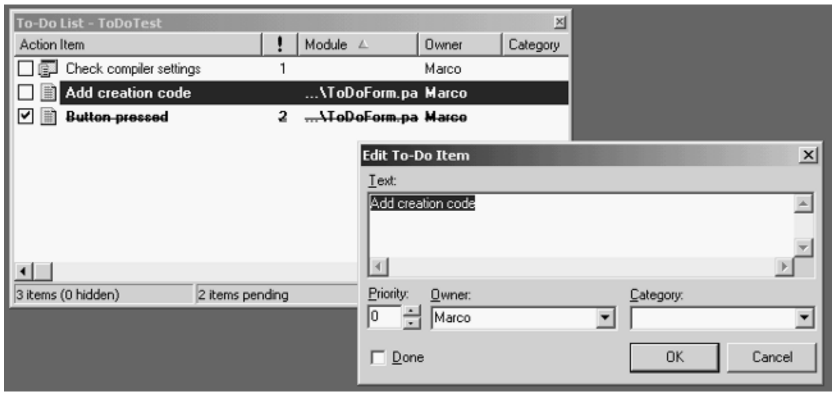
Shared repository
Directory:

FIGURE 1.2
The Preferences page of the Environment Options dialog box

You can add or modify to-do items by adding special `TODO` comments to the source code of any file of a project; you'll then see the corresponding entries in the list. In addition, you can visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.3, along with the To-Do List window.



Action Item	!	Module	Owner	Category
[ ] Check compiler settings	1		Marco	
[ ] Add creation code		...\ToDoForm.pa	Marco	
[✓] ~~Button pressed~~	~~2~~	~~...\ToDoForm.pa~~	~~Marco~~	

3 items (0 hidden) 2 items pending

Text: Add creation code
Priority: 0
Owner: Marco
Category:
[ ] Done

FIGURE 1.3
The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the same root name as the project file and a .TODO extension.

You can use multiple options with a `TODO` comment. You can use `–o` (as in the previous code excerpt) to indicate the owner (the programmer who entered the comment), the `–c` option to indicate a category, or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

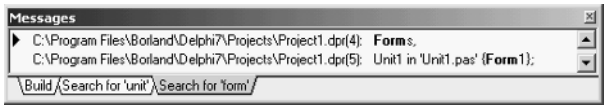
Delphi treats everything after the colon—up to the end of the line or the closing brace, depending on the type of comment—as the text of the to-do item.

Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from `TODO` to `DONE`. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, which can automatically collect to-do information from the source code files as you type them, sort and filter them, and export them to the Clipboard as plain text or an HTML table. All these options are available on the context menu.

Extended Compiler Messages and Search Results in Delphi 7

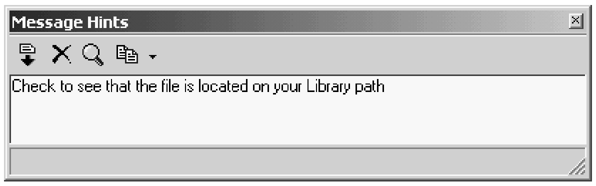
A small Messages window appears by default below the editor; it displays both compiler messages and search results. This window has been considerably modified in Delphi 7. First, search results are displayed in a different tab so they do not interfere with compiler messages as they did in the past. Second, every time you do a different search you can request that Delphi show the results in a different page, so the results of previous search operations remain available:



```
C:\Program Files\Borland\Delphi7\Projects\Project1.dpr(4):  Forms,
C:\Program Files\Borland\Delphi7\Projects\Project1.dpr(5):  Unit1 in 'Unit1.pas' {Form1};
```

You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the tabs of this window. (The same commands work for other tabbed views.)

If compiler errors occur, you can activate another new window with the command View ➢ Additional Message Info. As you compile a program, this Message Hints window will provide extra information for some common error messages, offering suggestions about how to fix them:



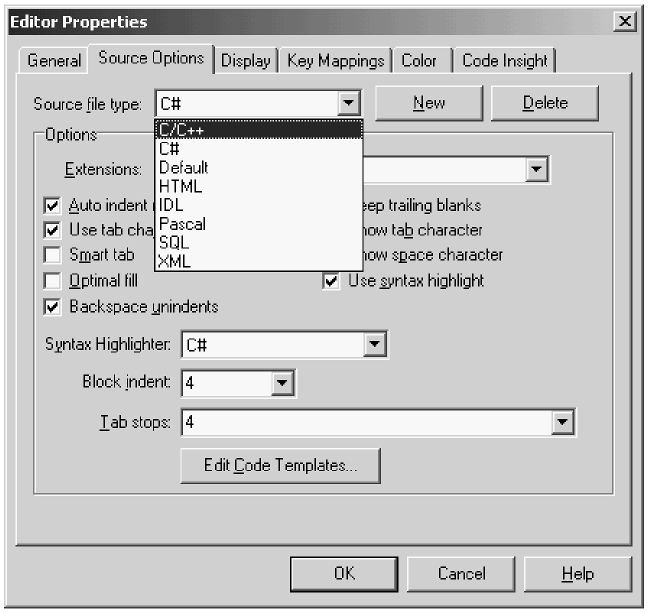
Check to see that the file is located on your Library path

This type of help is intended more for novice programmers, but it might be handy to keep this window around. It's important to realize that this information is thoroughly customizable: A project development leader can put appropriate descriptions of common errors in a form that means something specific to new developers. To do so, follow the comments in the file hosting the settings for this feature, the `msginfo70.ini` file of Delphi's `bin` folder.

The Delphi Editor

On the surface, Delphi's editor doesn't appear to have changed much for version 7 of the IDE. However, behind the scenes, it is a totally new tool. Besides using it to work on files in the Object Pascal language (or the Delphi language, as Borland prefers to call it now), you can now use it to work on other files used in Delphi development (such as SQL, XML, HTML, and XSL files), as well as files in other languages (including C++ and C#). XML and HTML editing was already available in Delphi 6, but the changes in this version are significant. For example, while editing an HTML file, you have support for both syntax highlighting and code completion.

The editor settings used on each file (including the behavior of keys like Tab) depend on the extension of the file being opened. You can configure these settings in the new Source Options page of the Editor Properties dialog box, displayed in Figure 1.4. This feature has been extended and made more open, so you can even configure the editor by providing a DTD for XML-based file formats or by writing a custom wizard that provides syntax highlighting for other programming languages. Another feature of the editor, code templates, is now language specific (your predefined Delphi templates will make little sense in HTML or C#).



Source file type: C#

Extensions:

- [✓] Auto indent
- [✓] Use tab cha
- [ ] Smart tab
- [ ] Optimal fill
- [✓] Backspace unindents
- ep trailing blanks
- ow tab character
- ow space character
- [✓] Use syntax highlight

Syntax Highlighter: C#

Block indent: 4

Tab stops: 4

FIGURE 1.4 The multiple languages supported by the Delphi IDE can be associated with various file extensions in the Source Options page of the Editor Properties dialog box.

NOTE *C# is the new language Microsoft introduced with its .NET architecture. Borland is expected to support C# in its own .NET environment, currently code-named Galileo.*

Considering only the Delphi language, the editor included in the IDE hasn't changed much in recent versions. However, it has a few features that many Delphi programmers don't know about and use, so I think it's worth a brief examination.

The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Ctrl+Shift+Tab to move in the opposite direction). You can drag-and-drop the tabs with the unit names in the upper portion of the editor to change their order, so that you can use a single Ctrl+Tab to move between the units you are working on any given time. The editor's shortcut menu has also a Pages command, which lists all the available pages in a submenu (a handy feature when many units are loaded).

You can also open multiple editor windows, each hosting multiple tabs. Doing so is the only way to see the source code of two units alongside each other. (Actually, when I need to compare two Delphi units, I invariably use Beyond Compare—`www.scootersoftware.com`—a superb, low-cost file comparison utility written in Delphi.)

Several options affect the editor, as you can see in the Editor Properties dialog box in Figure 1.4. However, you have to go to the Preferences page of the Environment Options dialog box (see Figure 1.2) to set the editor's AutoSave feature. This option forces the editor to save all of your source code files each time you run the program, preventing data loss in the (rare) case the program crashes badly in the debugger.

Delphi's editor provides many commands, including some that date back to its WordStar emulation ancestry (of the early Turbo Pascal compilers). I won't discuss the various settings of the editor, because they are quite intuitive and are described in the online help. Notice, though, that the page of the help describing the keyboard shortcuts is accessible as a whole only if you look up the *shortcuts* index item.

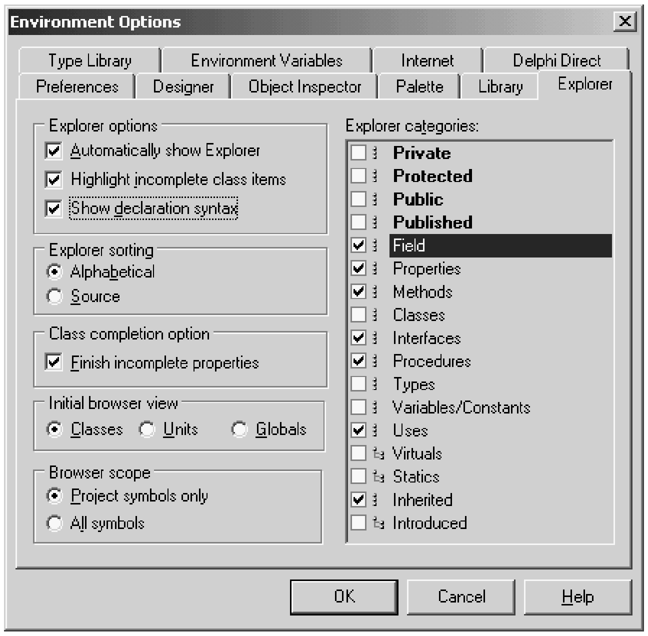
TIP *A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. In addition, you can copy text instead of moving it, by pressing the Ctrl key while dragging.*

The Code Explorer

The Code Explorer window, which is generally docked on the side of the editor, lists all the types, variables, and routines defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you begin typing in the editor.

You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration. You can also modify variables, properties, and method names directly in the Code Explorer. However, as you'll see, if you want a visual tool to use when you work on your classes, ModelMaker provides many more features.

Although all this functionality is obvious after you've used Delphi for a few minutes, some features of the Code Explorer are not so intuitive. You have full control of the information layout. And, you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer (collapsing the tree can help you make your selections more quickly). You can configure the Code Explorer by using the corresponding page of the Environment Options, as shown in Figure 1.5.



Explorer options
[✓] Automatically show Explorer
[✓] Highlight incomplete class items
[✓] Show declaration syntax

Explorer sorting
(●) Alphabetical
( ) Source

Class completion option
[✓] Finish incomplete properties

Initial browser view
(●) Classes ( ) Units ( ) Globals

Browser scope
(●) Project symbols only
( ) All symbols

Explorer categories:
[ ] **Private**
[ ] **Protected**
[ ] **Public**
[ ] **Published**
[✓] Field
[✓] Properties
[✓] Methods
[ ] Classes
[✓] Interfaces
[✓] Procedures
[ ] Types
[ ] Variables/Constants
[✓] Uses
[ ] Virtuals
[ ] Statics
[✓] Inherited
[ ] Introduced

FIGURE 1.5
You can configure the Code Explorer in the Environment Options dialog box.

Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view—it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer; on the contrary, the used units are listed as main nodes instead of being kept in the `Uses` folder. I generally disable the Types, Classes, and Variables/Constants selections.

Because each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, because this arrangement requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer provides a handy way of navigating the source code of a large unit—when you double-click a method in the Code Explorer, the focus moves to the definition in the class declaration. You can use Module Navigation (the Ctrl+Shift combination with the Up or Down arrow key) to jump from the definition of a method or procedure to its complete definition (or back again).

NOTE *Some of the Explorer Categories shown in Figure 1.5 are used by the Project Browser, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.*

Browsing in the Editor

Another feature of the editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the library.

WARNING *Although it may seem like a good idea at first, you cannot use Tooltip symbol insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, the Tooltip won't appear.*

The real bonus of this feature, however, is that you can turn it into a navigational aid called *code browsing*. When you hold down the Ctrl key and move the mouse over the identifier, Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of links in web browsers, and the pointer changes to a hand whenever it's positioned on the link.

For example, you can Ctrl+click the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again, as in a web browser—using the Browse Back and Browse Forward buttons in the top-right corner of the editor windows or the keystrokes Alt+Left arrow or Alt+Right arrow. You can also click the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

How can you jump directly to the VCL source code if it is not part of your project? The editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment.

Class Completion

Delphi's editor can also help by generating some source code for you, completing what you've already written. This feature is called *class completion*, and you activate it by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, because Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi simplify life in a similar way for programmers who write a little extra code behind event handlers. This code-generation feature applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }
procedure TForm1.Hello(MessageText: string);
begin
end;
```

This feature is really handy compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin...end` code for every method copied. Class completion also works the other way around: You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

The most important and useful example of class completion is the automatic generation of code to support properties declared in classes. For example, if you type in a class

```
property Value: Integer;
```

and press Ctrl+Shift+C, Delphi will turn the line into

```
property Value: Integer read fValue write SetValue;
```

Delphi will also add the `SetValue` method to the class declaration and provide a default implementation for it. You'll find more on properties in the next chapter.

Code Insight

In addition to the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing of both the source code you write and the source code of the system units your source code refers to.

Code insight comprises five capabilities: code completion, code templates, code parameters, Tooltip expression evaluation, and Tooltip symbol insight. This last feature was already covered in the section "Browsing in the Editor"; the other four are discussed in the following subsections. You can enable, disable, and configure each of these features in the Code Insight page of the Editor Properties dialog box.

Code Completion

Code completion allows you to choose the property or method of an object simply by looking it up on a list or by typing its initial letters. To activate this list, you just type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press Ctrl+spacebar; to remove it when you don't want it, press Esc. Code completion also lets you look for a proper value in an assignment statement.

As you begin typing, the list filters its content according to the initial portion of the element you've inserted. The code completion list uses colors and shows more details to help you distinguish different items. In Delphi, you can customize these colors in the Code Insight page of the Editor Options dialog box. Another feature is that in the case of functions with parameters, parentheses are included in the generated code, and the parameters list hint is displayed immediately.

As you type `:=` after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click it to change the order of the items, sorting either by scope or by name; you can also resize the window.

Since Delphi 6, code completion also works in the interface section of a unit. If you press Ctrl+spacebar while the cursor is inside the class definition, you'll get a list of virtual methods you can override (including abstract methods), the methods of implemented interfaces, the base class properties, and eventually system messages you can handle. Simply selecting one of them will add the proper method to the class declaration. In this particular case, the code completion list allows multiple selection.

TIP *When the code you've written is incorrect, code insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific code insight errors in the Message pane (which must already be open—it doesn't open automatically to display compilation errors). To activate this feature, you need to set an undocumented Registry entry, setting the string key* `\Delphi\7.0\Compiling\ShowCodeInsiteErrors` *to the value '1'.*

Code completion includes some advanced features that aren't easy to spot. One that I find particularly useful relates to the discovery of symbols in units not used by your project. As you invoke it (with Ctrl+spacebar) over a blank line, the list also includes symbols from common units (such as Math, StrUtils, and DateUtils) not already included in the `uses` statement of the current unit. By selecting one of these *external* symbols, Delphi adds the unit to the `uses` statement for you. This feature (which doesn't work inside expressions) is driven by a customizable list of extra units, stored in the Registry key `\Delphi\7.0\CodeCompletion\ExtraUnits`.

TIP *Delphi 7 adds the ability to browse to the declaration of items in the code completion list by Ctrl+clicking on any identifier in the list.*

Code Templates

This feature lets you insert one of the predefined code templates, such as a complex statement with an inner `begin...end` block. Code templates must be activated manually, by pressing Ctrl+J to show a list of all of the templates. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.

You can add custom code templates, so that you can build your own shortcuts for commonly used blocks of code. For example, if you use the `MessageDlg` function often, you might want to add a template for it. To modify templates, go to the Source Options page of the Editor Options dialog box, select Pascal from the Source File Type list, and click the Edit Code Templates button. Doing so opens the new Delphi 7 Code Templates dialog box. At this point, click the Add button, type in a new template name (for example, **mess**), type a description, and then add the following text to the template body in the Code memo control:

```
MessageDlg ('|', mtInformation, [mbOK], 0);
```

Now, every time you need to create a message dialog box, you simply type **mess** and then press Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position within the source code where the cursor will be in the editor after you expand the template. You should choose the position where you want to begin typing to complete the code generated by the template.

Although code templates might seem at first to correspond to language keywords, they are a more general mechanism. They are saved in the `DELPHI32.DCI` file, a text file in a rather simple format that you can edit directly. Delphi 7 also allows you to export the settings for a language to a file and import them, making it easier for developers to exchange their own customized templates.

Code Parameters

While you are typing a function or method, code parameters display the data type of the function's or method's parameters in a hint or Tooltip window. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a pop-up hint window. To force the display of code parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in bold type.

Tooltip Expression Evaluation

Tooltip expression evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor. In the case of an expression, you typically need to select it in the editor and then move the mouse over the highlighted text.

More Editor Shortcut Keys

The editor has many more shortcut keys that depend on the editor style you've selected. Here are a few of the lesser-known shortcuts:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark, press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the facts that a new bookmark can override an existing one and that bookmarks are not persistent (they are lost when you close the file).
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is set by the Block Indent option in the Editor page of the Editor Options dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R begins recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that might have little meaning once you've moved on to a different source code file. Editor macros are quite useful for performing multistep operations over and over again, such as reformatting source code or arranging data more legibly in source code.
- While holding down the Alt key, you can drag the mouse to select rectangular areas within the editor, not just consecutive lines and words.

Loadable Views

Another important feature introduced in Delphi 6 is support for multiple views in the editor. For any single file loaded in the IDE, the editor can show multiple views, defined programmatically and added to the system, and then loaded for given files—hence the name *loadable* views.

The most frequently used view is the Diagram page, which was available in Delphi 5 data modules, although it was less powerful. Another set of views is available in web applications, including an HTML Script view, an HTML Result preview, and many others discussed in Chapters 20 ("Web Programming with WebBroker and WebSnap") and 22 ("Using XML Technologies"). You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the bottom tabs of this editor; Ctrl+Tab changes the pages (or files) shown in the upper tabs.

The Diagram View

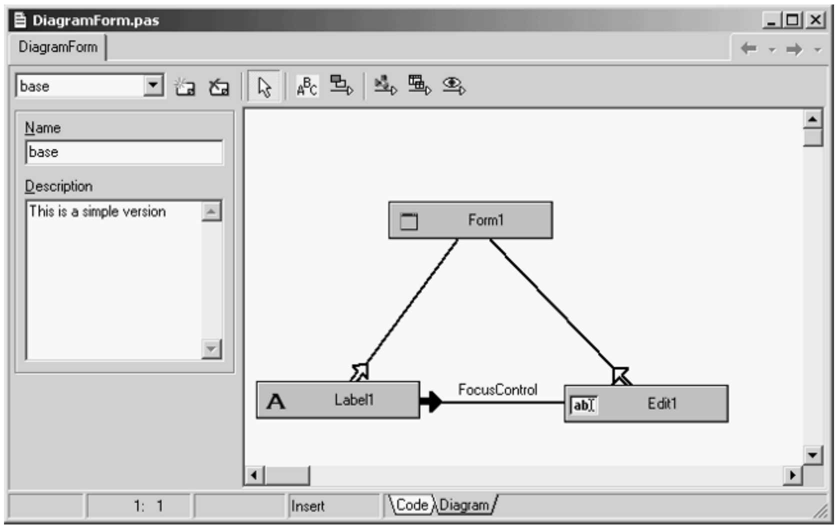
The Diagram view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections. You can even add your comments in text blocks linked to specific components.

The diagram is not built automatically. You must drag components from the Tree view to the diagram, which will automatically display the existing relations among the components you drop there. You can select multiple items from the Object TreeView and drag them all at once to the Diagram page.

What's nice is that you can set properties by simply drawing arrows between the components. For example, after moving an edit and a label to the diagram, you can select the Property Connector icon, click the label, and drag the mouse cursor over the edit. When you release the mouse button, the Diagram view will set up a property relation based on the `FocusControl` property, which is the only property of the label referring to an edit control. This situation is depicted in Figure 1.6.

Figure 1.6

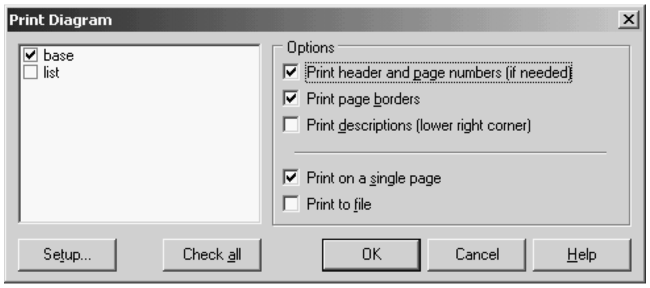
The Diagram view shows relationships among components (and even allows you to set them up).



As you can see, setting properties is *directional*: If you drag the property relation line from the edit to the label, you end up trying to use the label as the value of a property of the edit box. Because this isn't possible, you'll see an error message indicating the problem and offering to connect the components in the opposite way. The Diagram view allows you to create multiple diagrams for each Delphi unit—that is, for each form or data module. You give a name to the diagram and possibly add a description, click the New Diagram button, prepare another diagram, and you'll be able to switch back and forth between diagrams using the combo box available in the toolbar of the Diagram view.

Although you can use the Diagram view to set up relations, its main role is to document your design. For this reason, it is important to be able to print the content of this view. Use the standard File ➢ Print command while the Diagram view is active, and Delphi prompts you for options as shown in Figure 1.7. You can customize the output in many ways.

FIGURE 1.7
The Print Options for the Diagram view



The information in the Diagram view is saved in a separate file, not as part of the DFM file. Delphi 5 used design-time information (DTI) files, which had a structure similar to INI files. Delphi 6 and 7 can still read the older .DTI format, but they use the new Delphi Diagram Portfolio format (.DDP). These files use the DFM binary format (or a similar format), so they are not editable as text. All of these files are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

NOTE *If you want to experiment with the Diagram view, you can start by opening the DiagramDemo project included among the examples for this chapter. The program's form has two associated diagrams: the one in Figure 1.6 and a much more complex one with a pull-down menu and its items.*

The Form Designer

Another Delphi window you'll interact with often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you can select a component directly with the mouse; you can also use the Object Inspector or the Object TreeView, which is handy when a control is behind another one or is very small. If one control covers another completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click the selected component. Doing so will deselect the current component and select the form by default.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position (when the Snap To Grid option is active), as is holding down Alt while using the mouse to move the control. If you press Ctrl+Shift along with an arrow key, the component will move only at grid intervals.

By pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. Again, you can also do so with the mouse and the Alt key.

To align multiple components or make them the same size, you can select them and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, click them with the mouse while holding down the Shift key; or, if all the components fall into a rectangular area, drag the mouse to "draw" a rectangle surrounding them. To select child controls (say, the buttons inside a panel), drag the mouse within the panel while holding down the Ctrl key—otherwise, you move the panel. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment Palette (accessible through the View ➢ Alignment Palette menu command).

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component. This command is particularly helpful, because Undo operations on forms are limited (you can only Undelete), but the setting is not persistent.

Among its other features, the Form Designer offers several Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. Since version 6, Delphi offers extended hints, with details about the control's position, size, tab order, and more. This is an addition to the Show Component Captions environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

Finally, you can save DFM (Delphi Form Module) files in the old binary resource format, instead of the plain text format, which is the default. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

Having DFM files stored as text lets you operate more effectively with version-control systems. Programmers won't get a real advantage from this feature, because you could already open the binary DFM files in the Delphi editor with a specific command from the designer's shortcut menu. Version-control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file.

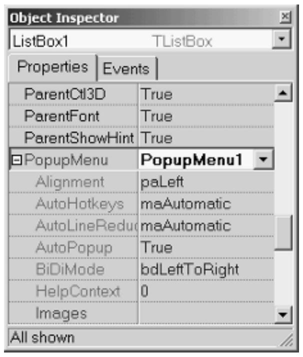
In any case, if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFMs are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded more quickly).

NOTE *Text DFM files are more portable between versions of Delphi than their binary version. Although an older version of Delphi might not accept a new property of a control in a DFM created by a newer version of Delphi, the older Delphis will still be able to read the rest of the text DFM file. If the newer version of Delphi adds a new data type, though, older Delphis will be unable to read the newer Delphi's binary DFMs at all. Even if this doesn't sound likely, remember that 64-bit operating systems are just around the corner. When in doubt, save in text DFM format. Also note that all versions of Delphi support text DFMs, using the command-line tool Convert in the* `bin` *directory. Finally, keep in mind that the CLX library uses the XFM extension instead of the DFM extension, both in Delphi and Kylix.*

The Object Inspector

To see and change properties of components placed on a form (or another designer) at design time, you use the Object Inspector. Compared to the early versions of Delphi the Object Inspector has a number of new features. The latest, introduced in Delphi 7, is the use of a bold font to highlight properties that have a value different from the default.

Another important change (introduced in Delphi 6) is the ability of the Object Inspector to expand component references in place. Properties referring to other components are displayed in a different color and can be expanded by selecting the + symbol on the left, as is the case with an internal subcomponent. You can then modify the properties of that other component without having to select it. Here you can see a connected component (a pop-up menu) expanded in the Object Inspector while working on another component (a list box):



ParentCtl3D	True
ParentFont	True
ParentShowHint	True
⊟PopupMenu	**PopupMenu1**
Alignment	paLeft
AutoHotkeys	maAutomatic
AutoLineRedu	maAutomatic
AutoPopup	True
BiDiMode	bdLeftToRight
HelpContext	0
Images	

This interface-expansion feature also supports subcomponents, as demonstrated by the new LabeledEdit control. A related feature of the Object Inspector lets you select the component referenced by a property. To do this, double-click the property value with the left mouse button while pressing the Ctrl key. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the Menu property of the form and Ctrl+double-clicking the value of this property. Doing so selects the main menu indicated as the value of the property in the Object Inspector.

Here are some other recent changes of the Object Inspector:

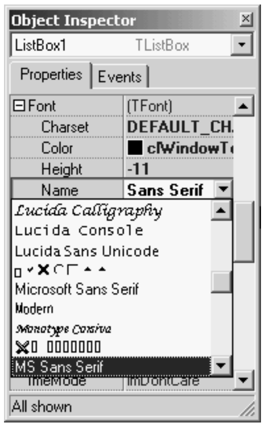
- The list at the top of the Object Inspector shows the type of the object and allows you to choose a component. You might remove this list to save some space, considering that you can select components in the Object TreeView (by default placed on the top of the Object Inspector window).
- The properties that reference an object are now a different color and may be expanded without changing the selection.
- You can optionally view read-only properties in the Object Inspector. Of course, they are grayed out.
- The Object Inspector has a Properties dialog box, which allows you to customize the colors of the various types of properties and the overall behavior of this window.
- Since Delphi 5, the drop-down list for a property can include graphical elements. This feature is used for properties such as `Color` and `Cursor`, and is particularly useful for the `ImageIndex` property of components connected to an ImageList.

NOTE *Interface properties can now be configured at design time using the Object Inspector. This functionality uses the Interfaced Component Reference model introduced in Kylix/Delphi 6, where components can implement and hold references to interfaces as long as the interfaces are implemented by components. Interfaced Component References work like plain old component references, but interface properties can be bound to any component that implements the necessary interface. Unlike component properties, interface properties are not limited to a specific component type (a class or its derived classes). When you click the drop-down list in the Object Inspector editor for an interface property, all components on the current form (and linked forms) that implement the interface are shown.*

Drop-Down Fonts in the Object Inspector

The Delphi Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the `Name` subproperty of the `Font` property. This capability is built into Delphi, but it has been disabled because most computers have a large number of fonts installed and rendering them can significantly slow the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplay-FontNames` global variable of the `VCLEditors` unit. I've done this in the OiFontPk package, which you can find among the program examples for this chapter.

Once this package is installed, you can move to the `Font` property of any component and use the graphical Name drop-down menu, as displayed here:



⊟ Font	(TFont)
Charset	DEFAULT_CH
Color	■ clWindowT
Height	-11
Name	Sans Serif

Lucida Calligraphy
Lucida Console
Lucida Sans Unicode
Microsoft Sans Serif
Modern
Monotype Corsiva
MS Sans Serif

There is a second, more complex customization of the Object Inspector that I like and use frequently: a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations. See Appendix A to learn how to obtain this add-on package.

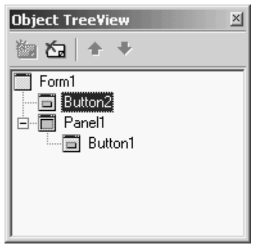
Property Categories

Delphi includes the idea of property categories, activated by the Arrange option of the Object Inspector's shortcut menu. If you set this option, properties are arranged by group rather than listed alphabetically, with each property possibly appearing in multiple groups. Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). Although property categories have been available since Delphi 5, programmers rarely use them.

The Object TreeView

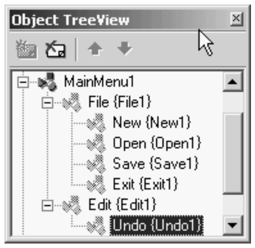
Delphi 5 introduced a TreeView for data modules, in which you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi 6 extended the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector.

The Object TreeView shows all the components and objects on the form in a tree representing their relations. The most obvious is the parent/child relation: If you place a panel on a form, a button inside the panel, and a button outside the panel, the tree will show one button under the form and the other under the panel:



Notice that the TreeView is synchronized with the Object Inspector and Form Designer. So, as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

Besides parent/child, the Object TreeView shows other relations, such as owner/owned, component/subobject, and collection/item, plus various specific relations, including dataset/connection and data source/dataset relations. Here, you can see an example of the structure of a menu in the tree:

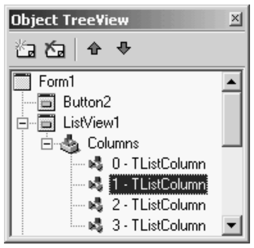


At times, the TreeView also displays "dummy" nodes, which do not correspond to an actual object but do correspond to a predefined object. As an example of this behavior, drop a Table component (from the BDE page); you'll see two grayed icons for the session and the alias. Technically, the Object TreeView uses gray icons for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects that are constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties. If you drop a Table on the form, you'll also see items that have next to them a red question mark enclosed in a yellow circle. This symbol indicates partially undefined items.

The Object TreeView supports multiple types of *dragging*:

- You can select a component from the palette (by clicking it, not dragging it), move the mouse over the tree, and click a component to drop it there. This technique allows you to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components—something that prevents you from dropping the component in the designer without first rearranging those components.
- You can drag components within the TreeView—for example, moving a component from one container to another. With the Form Designer, you can do so only with cut-and-paste techniques. Moving instead of cutting provides the advantage that any connections among components are not lost, as happens when you delete the component during the cut operation.
- You can drag components from the TreeView to the Diagram view, as you'll see later.

Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases, the shortcut menu may include items related to the custom component editors). You can even delete items from the tree. The TreeView also doubles as a collection editor, as you can see here for the `Columns` property of a ListView control. In this case, you can not only rearrange and delete items, but also add new items to the collection.



TIP *You can print the contents of the Object TreeView for documentation purposes. Simply select the window and use the File ➢ Print command (there is no Print command on the shortcut menu).*

Secrets of the Component Palette

The Component Palette is used to select components you want to add to the current designer. Move the mouse over a component and you'll see its name. In Delphi 7, the hint displays also the name of the unit that defines the component.

The Component Palette has many tabs—far too many, really. You may want to hide the tabs hosting components you don't plan to use and reorganize the Component Palette to suit your needs. In Delphi 7 you can also drag and drop the tabs to reorder them. Using the Palette page of the Environment Options dialog box, you can completely rearrange the components in the various pages, adding new elements or moving them from page to page.

When you have too many pages in the Component Palette, you'll need to scroll through them to reach a component. You can use a simple trick in this case: Rename the pages with shorter names, so all the pages will fit on the screen. (It's obvious—once you've thought about it.)

Delphi 7 offers another new feature. When there are too many components on a single page, Delphi displays a double down arrow; you click it to display the remaining components without having to scroll within the Palette page.

The Component Palette's shortcut menu has a Tabs submenu that lists all the palette pages in alphabetical order. You can use this submenu to change the active page, particularly when the page you need is not visible on the screen.

TIP *You can set the order of the entries in the Tabs submenu of the Component Palette shortcut menu to be the same as the order in the palette itself, rather than alphabetical. To do so, go to the* `Main Window` *Registry section of Delphi (under* `\Software\Borland\Delphi\7.0` *for the current user) and set the* `Sort Palette Tabs Menu` *key value to 0 (false).*

The significant undocumented feature of the Component Palette is "hot-track" activation. By setting special keys in the Registry, you can simply select a page of the palette by moving the mouse over the tab, without clicking. The same feature can be applied to the component scrollers on both sides of the palette, which appear when a page has too many components. To activate this hidden feature, add an `Extras` key under the `Borland\Delphi\7.0` key of the Registry in the `HKEY_CURRENT_USER\Software` section. Under this key, enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string '1'.

Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate a component in the form. During this operation, Delphi duplicates all the properties, keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

You can also copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source-code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
```

```
    Width = 75
    Height = 25
    Caption = 'My Button'
    TabOrder = 0
    Font.Name = 'Arial'
  end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

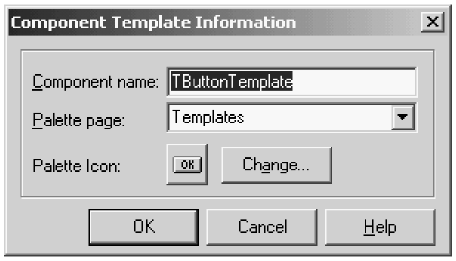
To use this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something that's completely incorrect and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This approach might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➢ Create Component Template menu command. This command opens the Component Template Information dialog box, where you enter the name of the template, the page of the Component Palette where it should appear, and an icon:



Component name: TButtonTemplate
Palette page: Templates
Palette Icon:

By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

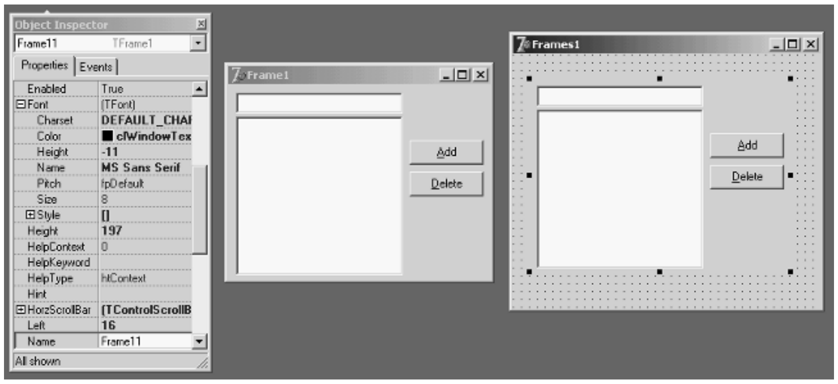
All the information about component templates is stored in a single file, `DELPHI32.DCT`, but there is no documented way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

TIP *A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry* `CCLibDir` *under the key* `\Software\Borland\Delphi\7.0\Component Templates`.

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready, you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each instance of the frame.

You can see a simple example, called Frames1, in Figure 1.8. A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames.



Enabled	True
⊟ Font	(TFont)
Charset	DEFAULT_CHAR
Color	■ clWindowTex
Height	-11
Name	MS Sans Serif
Pitch	fpDefault
Size	8
⊞ Style	[]
Height	197
HelpContext	0
HelpKeyword	
HelpType	htContext
Hint	
⊞ HorzScrollBar	(TControlScrollB
Left	16
Name	Frame11

FIGURE 1.8
The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

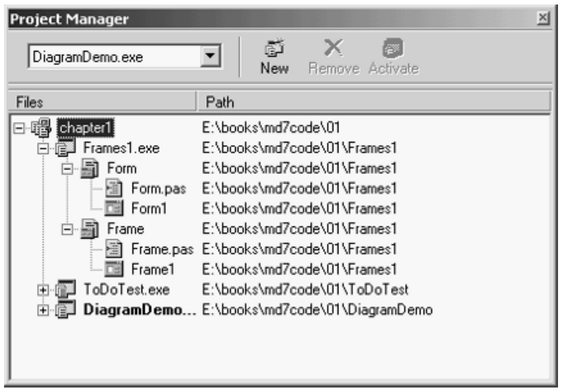
Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than component templates. Chapter 8, "The Architecture of Delphi Applications," provides an in-depth look at VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful technique.

Managing Projects

Delphi's multitarget Project Manager (View ➢ Project Manager) works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files. All open packages will show up as projects in the Project Manager view, even if they haven't been added to the project group.

In Figure 1.9, you can see the Project Manager with a simple project group, including all the examples of the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on the projects in the group. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, and to open a unit.

FIGURE 1.9
Delphi's multitarget Project Manager



```
Files                    Path
⊟ chapter1               E:\books\md7code\01
  ⊟ Frames1.exe          E:\books\md7code\01\Frames1
    ⊟ Form               E:\books\md7code\01\Frames1
        Form.pas         E:\books\md7code\01\Frames1
        Form1            E:\books\md7code\01\Frames1
    ⊟ Frame              E:\books\md7code\01\Frames1
        Frame.pas        E:\books\md7code\01\Frames1
        Frame1           E:\books\md7code\01\Frames1
  ⊞ ToDoTest.exe         E:\books\md7code\01\ToDoTest
  ⊞ DiagramDemo...       E:\books\md7code\01\DiagramDemo
```

TIP *Since Delphi 6, the Project Manager also shows the open packages, even if they haven't been added to the project group. A* package *is a collection of components or other units compiled to a separate executable file, as you'll discover in Chapter 10 ("Libraries and Packages").*

Of all the projects in the group, only one is active; this is the project you operate on when you select a command such as Project ➢ Compile. The Project menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.) When you have multiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

TIP *In Delphi 7, the Project Manager local menu allows you to compile projects beginning with a given one, using the Make All From Here or Build All From Here command. If you use this command on the first project of the group, you obtain the same effect as a Compile All or Build All command from the shortcut menu.*

Among the Project Manager's advanced features, you can drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project (drag-and-drop is also supported to open files in the code editor). You can easily see which project is selected and change it by using the combo box at the top of the Project Manager window, or by using the drop-down arrow next to the Run button on the Delphi toolbar.

Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu item, and choose Resource File (*.rc) as the file type. This resource file will be bound to the project automatically, even without a corresponding `$R` directive.

Delphi saves the project groups with the .BPG extension, which stands for Borland Project Group. This feature comes from C++Builder and from past Borland C++ compilers; this history is clearly visible when you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment. Here is a simple example:

```
#------------------------------
VERSION = BWS.01
#------------------------------
!ifndef ROOT
ROOT = $(MAKEDIR)\..
!endif
#------------------------------
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#------------------------------
PROJECTS = Project1.exe
#------------------------------
default: $(PROJECTS)
#------------------------------
Project1.exe: Project1.dpr
  $(DCC)
```

Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. Instead, you can invoke the Project Options dialog from the Project Manager for each project. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{$`*X*`+}` and `{$`*X*`-}` directives, where you replace *X* with the option you want to set. This second approach is more flexible, because it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All project options are saved automatically with the project, but in a separate file with a .DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command-line compilation. The two files have similar content but a different format: The *dcc* command-line compiler cannot use .DOF files, but needs the .CFG format.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This key combination inserts, at the top of the current unit, compiler directives that correspond to the current project options (including all of the new compiler warning settings), as in the following listing:

```
{$A8,B-,C+,D+,E-,F-,G+,H+,I+,J-,K-,L+,M-,N+,O+,P+,Q-,R-,S-,T-,U-,V+,W-,X+,Y+,Z1}
{$MINSTACKSIZE $00004000}
{$MAXSTACKSIZE $00100000}
{$IMAGEBASE $00400000}
{$APPTYPE GUI}
{$WARN SYMBOL_DEPRECATED ON}
{$WARN SYMBOL_LIBRARY ON}
{$WARN SYMBOL_PLATFORM ON}
{$WARN UNIT_LIBRARY ON}
{$WARN UNIT_PLATFORM ON}
{$WARN UNIT_DEPRECATED ON}
{$WARN HRESULT_COMPAT ON}
{$WARN HIDING_MEMBER ON}
{$WARN HIDDEN_VIRTUAL ON}
{$WARN GARBAGE ON}
{$WARN BOUNDS_ERROR ON}
{$WARN ZERO_NIL_COMPAT ON}
{$WARN STRING_CONST_TRUNCED ON}
{$WARN FOR_LOOP_VAR_VARPAR ON}
{$WARN TYPED_CONST_VARPAR ON}
{$WARN ASG_TO_TYPED_CONST ON}
{$WARN CASE_LABEL_RANGE ON}
{$WARN FOR_VARIABLE ON}
{$WARN CONSTRUCTING_ABSTRACT ON}
{$WARN COMPARISON_FALSE ON}
{$WARN COMPARISON_TRUE ON}
{$WARN COMPARING_SIGNED_UNSIGNED ON}
{$WARN COMBINING_SIGNED_UNSIGNED ON}
{$WARN UNSUPPORTED_CONSTRUCT ON}
{$WARN FILE_OPEN ON}
{$WARN FILE_OPEN_UNITSRC ON}
{$WARN BAD_GLOBAL_SYMBOL ON}
{$WARN DUPLICATE_CTOR_DTOR ON}
{$WARN INVALID_DIRECTIVE ON}
{$WARN PACKAGE_NO_LINK ON}
{$WARN PACKAGED_THREADVAR ON}
{$WARN IMPLICIT_IMPORT ON}
{$WARN HPPEMIT_IGNORED ON}
{$WARN NO_RETVAL ON}
{$WARN USE_BEFORE_DEF ON}
{$WARN FOR_LOOP_VAR_UNDEF ON}
{$WARN UNIT_NAME_MISMATCH ON}
{$WARN NO_CFG_FILE_FOUND ON}
```

```
{$WARN MESSAGE_DIRECTIVE ON}
{$WARN IMPLICIT_VARIANTS ON}
{$WARN UNICODE_TO_LOCALE ON}
{$WARN LOCALE_TO_UNICODE ON}
{$WARN IMAGEBASE_MULTIPLE ON}
{$WARN SUSPICIOUS_TYPECAST ON}
{$WARN PRIVATE_PROPACCESSOR ON}
{$WARN UNSAFE_TYPE OFF}
{$WARN UNSAFE_CODE OFF}
{$WARN UNSAFE_CAST OFF}
```

Compiling and Building Projects

There are several ways to compile a project. If you run the project (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select Project ➢ Build All instead, every file is compiled, even if it has not changed. You should only need this second command infrequently, because Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options, in which case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi Compiled Unit (DCU). (This step is performed only if the DCU file is not already up to date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

WARNING *Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows a strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.*

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't run unless the proper DLLs (such as `vcl70.bpl`, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course, if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 10.

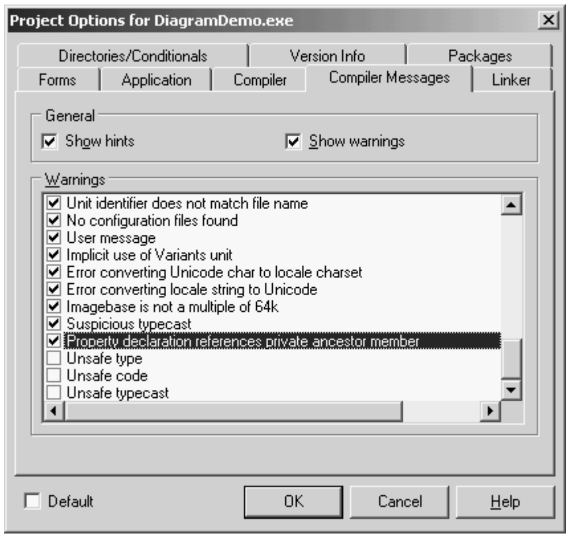
In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicompiled" tools.

Compiler Message Helpers and Warnings

As I mentioned at the beginning of this chapter (in the section "Extended Compiler Messages and Search Results in Delphi 7"), in addition to the classic compiler messages, Delphi 7 provides a new window with additional information about some error messages. This window is activated using the View ➢ Additional Message Info menu command. It displays information stored in a local file, which can be updated by downloading a new version from Borland's website.

Another change in Delphi 7 relates to the increased control you have over compiler warnings. The Project Options dialog box now includes a Compiler Messages page where you can choose many individual warnings. This feature was probably introduced due to the fact that Delphi 7 has a new set of warnings related to compatibility with the future Delphi for .NET tool. These warnings are quite extensive, and I've disabled them as shown in Figure 1.10.

FIGURE 1.10
The new Compiler Messages page of the Project Options dialog box



General
- [✓] Show hints
- [✓] Show warnings

Warnings
- [✓] Unit identifier does not match file name
- [✓] No configuration files found
- [✓] User message
- [✓] Implicit use of Variants unit
- [✓] Error converting Unicode char to locale charset
- [✓] Error converting locale string to Unicode
- [✓] Imagebase is not a multiple of 64k
- [✓] Suspicious typecast
- [✓] Property declaration references private ancestor member
- [ ] Unsafe type
- [ ] Unsafe code
- [ ] Unsafe typecast

[ ] Default

You can also enable or disable some of these warnings using compiler options like these:

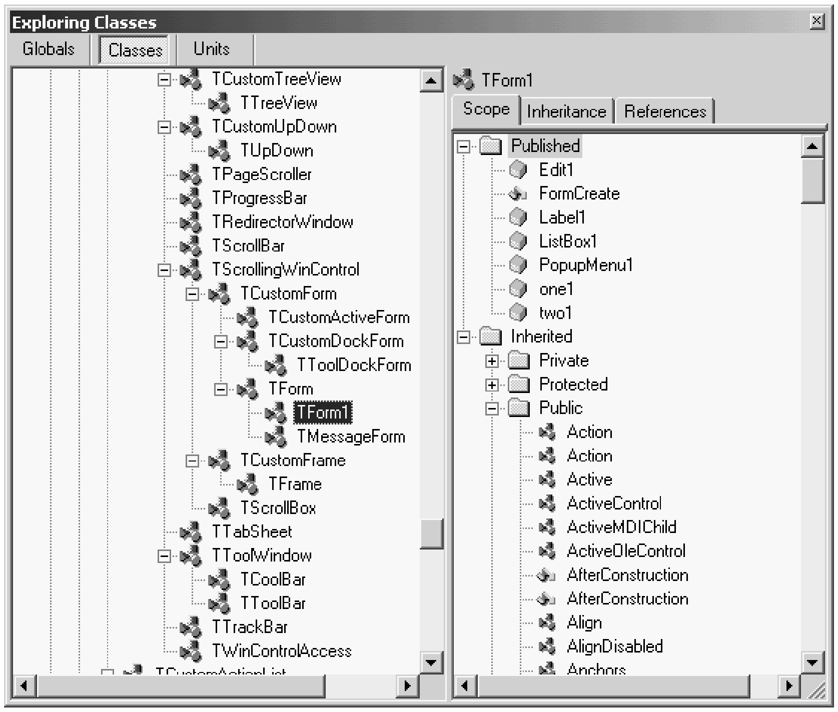
```
{$Warn UNSAFE_CODE OFF}
{$Warn UNSAFE_CAST OFF}
{$Warn UNSAFE_TYPE OFF}
```

In general, it is better to keep these settings outside the source code of the program—something Delphi 7 finally allows you to do.

Exploring a Project's Classes

Delphi has always included a tool to browse the symbols of a compiled project, although this tool's name has changed many times (from Object Browser to Project Explorer and now to Project Browser). In Delphi 7, you activate the Project Browser window using the View ➢ Browser menu command, which displays the window shown in Figure 1.11. The browser allows you to see the hierarchical structure of the project's classes and to look for its symbols and the source-code lines where they are referenced.

FIGURE 1.11
The Project Browser



Unlike the Code Explorer, the Project Browser is updated only as you recompile the project. This browser allows you to list classes, units, and globals, and lets you choose whether to look only for symbols defined within your project or for those from both your project and VCL. You can change the settings of the Project Browser and those of the Code Explorer in the Explorer page of the Environment Options or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the categories you see in this window are specific to the Project Browser; others relate to both tools.

Additional and External Delphi Tools

In addition to the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (`PCE.exe`), and the Image Editor (`ImagEdit.exe`), are available from the Tools menu in the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (`SqlMon.exe`). Other tools that are not directly accessible from the IDE include many command-line utilities you can find in the Delphi `bin` directory. For example, these tools include a command-line Delphi compiler (`DCC32.exe`), a Borland resource compiler (`BRC32.exe` and `BRCC32.exe`), and an executable viewer (`TDump.exe`).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools, most of which are available in the `\Delphi7\bin` folder and in the Tools menu:

Web App Debugger (`WebAppDbg.exe`) The debugging web server introduced in Delphi 6. It is used to keep track of the requests sent to your applications and to debug them. This debugger was rewritten in Delphi 7: It is now a CLX application and its connectivity is based on sockets. I'll discuss this tool in Chapter 20.

XML Mapper (`XmlMapper.exe`) A tool for creating XML transformations to be applied to the format produced by the ClientDataSet component. You'll find more on this topic in Chapter 22.

External Translation Manager (`etm60.exe`) The stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators and was available for the first time in Delphi 6.

Borland Registry Cleanup Utility (`D7RegClean.exe`) A tool that helps you remove all the Registry entries that Delphi 7 adds to a computer.

TeamSource An advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi. Delphi 7 ships with version 1.01 of Team Source, the same version available after applying an available patch to the Delphi 6 version.

WinSight (`Ws32.exe`) A Windows "message spy" program available in the `bin` directory.

Database Explorer A tool that can be activated from the Delphi IDE or as a stand-alone tool, using the `DBExplor.exe` program of the `bin` directory. Because it is meant for the BDE, the Database Explorer is not used much nowadays.

OpenHelp (`oh.exe`) The tool you can use to manage the structure of Delphi's own Help files, integrating third-party files into the help system.

Convert (`Convert.exe`) A command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

Turbo Grep (`Grep.exe`) A command-line search utility, which is much faster than the embedded Find In Files mechanism but not as easy to use.

Turbo Register Server (`TRegSvr.exe`) A tool you can use to register ActiveX libraries and COM servers. The source code for this tool is available under `\Demos\ActiveX\TRegSvr`.

Resource Explorer A powerful resource viewer (but not a full-blown resource editor) you can find under `\Demos\ResXplor`.

Resource Workshop An old 16-bit resource editor that can also manage Win32 resource files. The Delphi installation CD includes a separate installation for Resource Workshop. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS

EXTENSION	FILE TYPE AND DESCRIPTION	CREATION TIME	REQUIRED TO COMPILE?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to endusers.
.CAB	The Microsoft Cabinet compressed-file format used for web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	**FILE TYPE AND DESCRIPTION**	**CREATION TIME**	**REQUIRED TO COMPILE?**
.DCP	Delphi Compiled Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files or in the BPL file.	Compilation	Required when you use run-time packages. You'll distribute it only to other developers along with BPL files. You can compile an application with the units from a package just by having the DCP file and the BPL (and no DCU files).
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic link library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK and now also .DPKW and .DPKL	Delphi Package: the project source code file of a package (or a specific project file for Windows or Linux).	Development	Yes.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	**FILE TYPE AND DESCRIPTION**	**CREATION TIME**	**REQUIRED TO COMPILE?**
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (.DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	FILE TYPE AND DESCRIPTION	CREATION TIME	REQUIRED TO COMPILE?
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.
.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).

Besides the files generated during the development of a project in Delphi, many others are generated and used by the IDE itself. In Table 1.2, I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

TABLE 1.2: SELECTED DELPHI IDE CUSTOMIZATION FILE EXTENSIONS

EXTENSION	FILE TYPE
.DCI	Delphi code templates.
.DRO	Delphi's Object Repository. (The repository should be modified with the Tools ➢ Repository command.)
.DMT	Delphi menu templates.

Continued on next page

TABLE 1.2: SELECTED DELPHI IDE CUSTOMIZATION FILE EXTENSIONS *(continued)*

EXTENSION	FILE TYPE
.DBI	Database Explorer information.
.DEM	Delphi edit mask (files with country-specific formats for edit masks).
.DCT	Delphi component templates.
.DST	Desktop settings file (one for each desktop setting you've defined).

Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time covering their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is a single file for the form's whole code, not just small code fragments.

TIP *In the listings in this book, I've matched the bold syntax highlighting of the editor for keywords and the italic for strings and comments.*

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is by default a text representation of the form, but it can also be saved in a binary Windows Resource format. You can set the format you want to use for new projects in the Designer page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command on a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This command closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can edit the textual description of a form, although you should do so with extreme care. As soon as you save the file, it will be parsed to regenerate the form. If you've made incorrect changes, compilation will stop with an error message; you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have good knowledge of Delphi programming.

TIP *In the book, I often show you excerpts of DFM files. In most of these excerpts, I show only the relevant components or properties; generally, I have removed the positional properties, the binary values, and other lines providing little useful information.*

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application: the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the Project ➢ View Source menu command.

Some of the other, less relevant files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF):

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File ➢ New menu and in other pull-down menus. If you simply select File ➢ New ➢ Other, Delphi opens the Object Repository. You can use it to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New Items dialog box (shown in Figure 1.12) has several pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

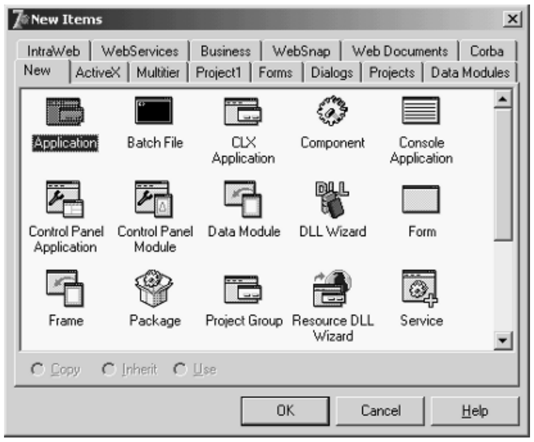


FIGURE 1.12
The first page of the New Items dialog box, generally known as the Object Repository

TIP *The Object Repository has a shortcut menu you can use to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool—information that is particularly important when you're looking at wizards, projects, or forms that you've added to the Repository.*

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, and it will be ready to use later. Simply use the Project ➢ Add To Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Move to the form that you want to add, and select the Add To Repository command from its shortcut menu. Then, indicate the title, description, author, page, and icon in the resulting dialog box. Keep in mind that as you copy a project or form template to the Repository and then copy it back to another directory, you are simply doing a copy-and-paste operation; this isn't much different than copying the files manually.

THE EMPTY PROJECT TEMPLATE

When you start a new project, Delphi automatically opens a blank form, too. However, if you want to base a new project on one of the form objects or wizards, you don't need this form. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1. Create a new project as usual.
2. Remove the project's only form.
3. Add this project to the templates, naming it *Empty Project*.

Continued on next page

The Empty Project Template *(continued)*

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File ➢ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➢ Repository command. This command opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing elements. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File ➢ New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File ➢ New Application), if no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File ➢ New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` subdirectory of the Delphi main directory. At the same time, if you use a form or any other object directly without copying it, then you end up having some of your project files in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files without copying data back and forth to the Repository.

Installing New DLL Wizards

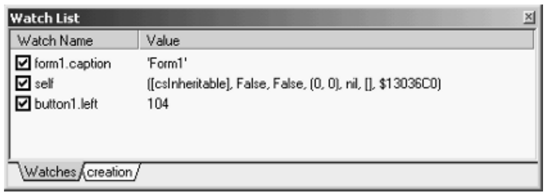
Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they are installed the same way you install a component or a package, using the Components ➢ Install Packages menu command and then clicking the Add button. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key `\Software\Borland\Delphi\7.0\Experts`. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

Debugger Updates in Delphi 7

When you run a program in Delphi's IDE, you generally start it in the integrated debugger. You can set breakpoints, execute the code line by line, and explore its inner details, including the assembler code being executed and the use of CPU registries in the CPU view. I don't have space in this book to cover debugging in Delphi; see Appendix C for information about extra material on this topic. However, I do want to briefly mention a couple of new debugger features.

First, the Run Parameters dialog box in Delphi 7 allows you to set a working directory for the program being debugged. This means the current folder will be the one you indicate, not the one the program has been compiled into.

Another relevant change relates to the Watch List. It now has multiple tabs that let you keep different sets of active watches for different areas of the program you are debugging, without cluttering a single window. You can add a new group to the Watch List by using its shortcut menu, and also alter the visibility of column headers and enable individual watches with the corresponding check boxes.



Watch Name	Value
✓ form1.caption	'Form1'
✓ self	([csInheritable], False, False, (0, 0), nil, [], $13036C0)
✓ button1.left	104

NOTE *This book doesn't cover the Delphi debugger, but this is a very important topic. See the references to online material in Appendix C for information about how to download a free chapter that discusses debugging in Delphi.*

What's Next?

This chapter has presented an overview of the new and more advanced features of the Delphi 7 programming environment, including tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to begin *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

Now we are ready to spend the next chapter looking into the Delphi programming language. Then we'll proceed by studying the run-time library (RTL) and the class library included in Delphi.