MARCO CANTÙ'S ESSENTIAL DELPHI

A Friendly Introductory Guide to Borland Delphi

http://www.marcocantu.com/edelphi

Revision 1.03 - April 13, 2002

INTRODUCTION

After the successful publishing of the e-book Essential Pascal (available on my web site at the address `http://www.marcocantu.com/epascal`), I decided to follow up with an introduction to Delphi. Again most of the material you'll find here was in the first editions of my "printed" book Mastering Delphi, the best selling Delphi book I have written. Due to space constraints and because many Delphi programmers look for more advanced information, in the latest edition this material was completely omitted. To overcome the absence of this information, I have started putting together this second on-line book, titled ***Essential Delphi***.

Copyright

Book Structure

The book structure is still under development, as the book evolves. This is the current structure:

Source Code

The source code of all the examples mentioned in the book is available on the book web site. The code has the same Copyright as the book: Feel free to use it at will but don't publish it on other documents or site. Links back to this site are welcome.

Feedback

Please let me know of any errors you find (indicating revision number and page number), but also of topics not clear enough for a beginner. I'll be able to devote time to the project depending also on the feedback I receive. Let me know also which other topics (not covered in Mastering Delphi) you'd like to see here.

For reporting errors please use the books section of my newsgroup, as described on www.marcocantu.com or use my mailbox (which gets far too jammed) at marco@marcocantu.com.

Acknowledgments

I have first started thinking about on-line publishing after Bruce Eckel's experience with *Thinking in Java*. I'm a friend of Bruce and think he really did a great job with that book and few others. After the overwhelming response of the "Essential Pascal" book, I started this new one and plan releasing the two as a printed book introducing Delphi (the only problem being to find a publisher).

About the Author

Marco Cantù lives in Piacenza, Italy. After writing C++ and Object Windows Library books and articles, he delved into Delphi programming. He is the author of the *Mastering Delphi* book series, published by Sybex, as well as the advanced *Delphi Developers Handbook*. He writes articles for many magazines, including *The Delphi Magazine*, speaks at Delphi and Borland conferences around the world, and teaches Delphi classes at basic and advanced levels. More recently, he's specializing in XML technologies, still making most of his programming in Delphi. Of course, you can learn more details about Marco and his work by visiting his web site, www.marcocantu.com.

Donations

I'll probably set up an account on one of those donation/contribution systems, to let people who have enjoyed the book and learned from it, particularly if programming is their job (and not a hobby) and they do it for profit, contribute to its development. No extra material is offered to those donating to the book fund, only because I want to let anyone (particularly students and people leaving in poor countries) benefit from the availability of this material. Information will be available on the book web site.

Table of Contents

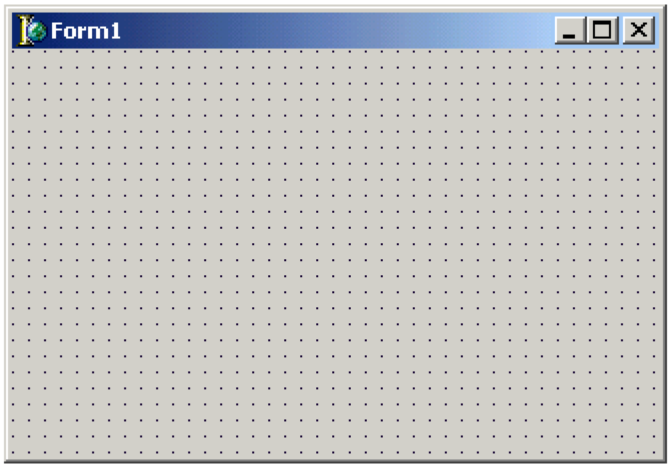
Chapter 1: A Form Is a Window

Windows applications are usually based on windows. So, how are we going to create our first window? We'll do it by using a form. As the first part of the title suggests, a form really is a window in disguise. There is no real difference between the two concepts, at least from a general point of view.

> If you look closely, a form is always a window, but the reverse isn't always true. Some Delphi components are windows, too. A push button is a window. A list box is a window. To avoid confusion, I'll use the term *form* to indicate the main window of an application or a similar window and the term *window* in the broader sense.

Creating Your First Form

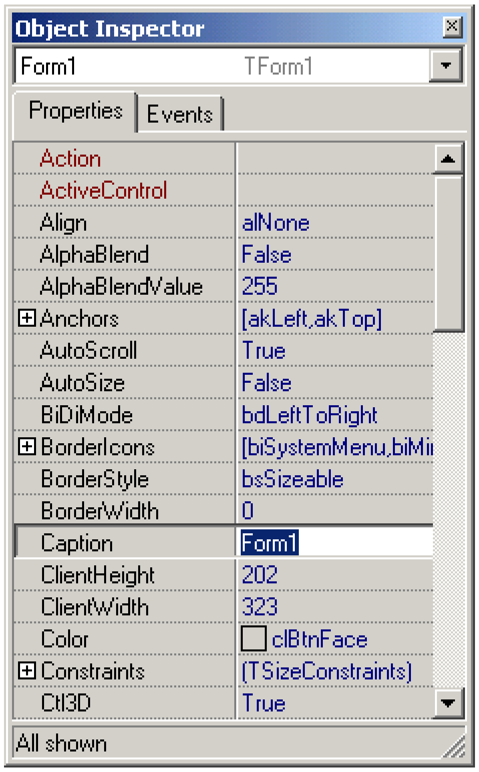
Even though you have probably already created at least some simple applications in Delphi, I'm going to show you the process again, to highlight some interesting points. Creating a form is one of the easiest operations in the system: you only need to open Delphi, and it will automatically create a new, empty form for you, as you can see in the figure below. That's all there is to it.



If you already have another project open, choose File | New | Application to close the old project (you may be prompted to save some of the files) and open a new blank project. Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with no capabilities, but the default behavior of any Windows window.

Adding a Title

Before we run the application, let's make a quick change. The title of the form is *Form1*. For a user, the title of the main window stands for the name of the application. Let's change *Form1* to something more meaningful. When you first open Delphi, the Object Inspector window should appear on the left side of the form (if it doesn't, open it by choosing View | Object Inspector or pressing the F11 key):



Action	
ActiveControl	
Align	alNone
AlphaBlend	False
AlphaBlendValue	255
⊞Anchors	[akLeft,akTop]
AutoScroll	True
AutoSize	False
BiDiMode	bdLeftToRight
⊞BorderIcons	[biSystemMenu,biMi
BorderStyle	bsSizeable
BorderWidth	0
Caption	Form1
ClientHeight	202
ClientWidth	323
Color	☐ clBtnFace
⊞Constraints	(TSizeConstraints)
Ctl3D	True

The Object Inspector shows the properties of the selected component. The window contains a tab control with two pages. The first page is labeled Properties. The other page is labeled Events and shows a list of events that can take place in the form or in the selected component.

The properties are listed in alphabetical order, so it's quite easy to find the ones you want to change (it is also possible to group them by category, as we'll see in the next chapter, but this feature is seldom used by Delphi developers). We can change the title of the form simply by changing the `Caption` property, which is selected by default. While you type a new caption, you can see the title of the form change. If you type *Hello*, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. If you have not entered a new caption, the new value of the `Name` property will be used for the `Caption` property, too.

Only a few of the properties of a component change while you type the new value. Most are applied when you finish the editing operation and press the Enter key (or move the input focus to a new property).

Although we haven't done much work, we have built a full-blown application, with a system menu and the default Minimize, Maximize, and Close buttons. You can resize the form by dragging its borders, move it by dragging its caption, maximize it to full-screen size, or minimize it. It works, but again, it's not very useful. If you look at the icon in the Taskbar, you'll see that something isn't right. Instead of showing the caption of the form as the icon caption, it shows the name of the project, something like *Project1*. We can fix this by giving a name to the project, which we'll do by saving it to disk with a new name.

Saving the Form

Select the Save Project or Save Project As command from the File menu, and Delphi will ask you to give a name to the source code file associated with the form, and then to name the project file. Since the name of the project should match the caption of the form (*Hello*), I've named the form source file `HELLOF.PAS`, which stands for *Hello Form*. I've given the project file the name `HELLO.DPR`.

Unfortunately, we cannot use the same name for the project and the unit that defines the form; for each application, these items must have unique names. You can add the letter *F*, add *Form*, call every form unit *MainForm*, or choose any other naming convention you like. I tend to use a name similar to the project name, as simply calling it `Mainform` means you'll end up with a number of forms (in different projects) that all have the same name.

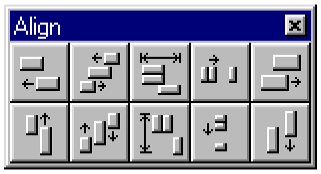
The name you give to the project file is used by default at run-time as the title of the application, displayed by Windows in the taskbar while the program is running. For this reason, if the name of the project matches the caption of the main form, it will also correspond to the name on the taskbar. You can also change the title of the application by using the Application page of the Project Options dialog box (choose Project | Options), or by writing a line of code to change the `Title` property of the Application global object.

Using Components

Now it's time to start placing something useful in our Hello form. Forms can be thought of as component containers. Each form can host a number of components or controls. You can choose a component from the Components Palette above the form, in the Delphi window. There are four simple ways to place a component on a form. If you choose the Button component from the Standard page of the Components Palette, for example, you can do any of the following:

- Click on the component, move the mouse cursor to the form, press the left mouse button to set the upper-left corner of the button, and drag the mouse to set the button's size.
- Select the component as above, and then simply click on the form to place a button of the default height and width.
- Double-click on the icon in the Components Palette, and a component of that type will be added in the center of the form.
- Shift-click on the component icon, and place several components of the same kind in the form using one of the above procedures.

Our form will have only one button, so we'll center it in the form. You can do this by hand, with a little help from Delphi. When you choose View | Alignment Palette, a toolbox with alignment icons appears:



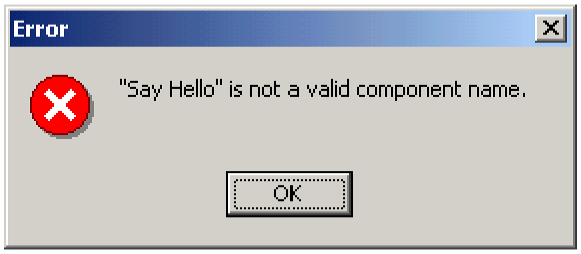
This toolbox makes a number of operations easy. It includes buttons to align controls or to center them in the form. Using the two buttons in the third column, you can place a component in the center of the form. Although we've placed the button in the center, as soon as you run the program, you can resize the form so that the button won't be in the center anymore. So the button is only in the center of the form at startup. Later on, we'll see how to make the button remain in the center after the form is resized, by adding some code. For now, our first priority is to change the button's label.

Changing Properties

Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you change the name of a button before changing its caption, the `Caption` property will have the same text as the `Name` property. Changing the `Name` property is usually a good choice, and you should generally do this early in the development cycle, before you write much code.

It is quite common to define a naming convention for each type of component (usually the full name or a shorter version, such as "btn" for Button). If you use a different prefix for each type of component (as in "*ButtonHello*" or "*BtnHello*"), the combo box above the Object Inspector will list the components of the same kind in a group, because they are alphabetically sorted. If you instead use a suffix, naming the components "*HelloButton*" or "*HelloBtn*," components of the same kind will be in different positions on the list. In this second case, however, finding a particular component using the keyboard might be faster. In fact, when the Object Inspector is selected you can type a letter to jump to the first component whose name starts with that letter.

Besides setting a proper name for a component, you often need to change its `Caption` property. There are at least two reasons to have a caption different from the name. The first is that the name often follows a naming convention (as described in the note above) that you won't want to use in a caption. The second reason is that captions should be descriptive, and therefore they often use two or more words, as in my *Say hello* button. If you try to use this text as the `Name` property, however, Delphi will show an error message:

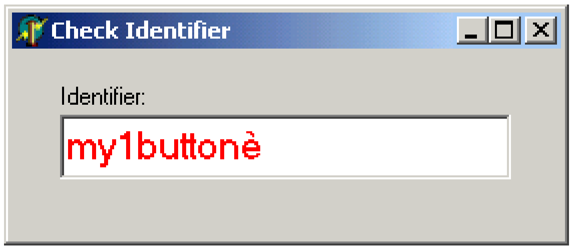

Error

"Say Hello" is not a valid component name.

The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the `Name` property, you must follow the rules for naming an identifier in the Pascal language:

- An identifier is a sequence of letters, digits, or underscore characters of any length (although only the first 63 characters are significant).
- The first character of an identifier cannot be a number; it must be a letter or the underscore character.
- No spaces are allowed in an identifier.
- Identifiers are not case-sensitive, but usually each word in an identifier begins with a capital letter, as in BtnHello. But btnhello, btnHello, and BTNHello refer to this same identifier.

You can use the `IsValidIdent` system function to check whether a given string is a valid identifier. The CheckId example calls this function while you type an identifier in its edit box, and changes the text color to indicate whether the string is valid (green) or not (red). The code of the example is quite simple, and you can look at it yourself on the disk. Try running this program to check any doubts about allowed component names.


Check Identifier

Identifier: my1buttonè

Here is a summary of the changes we have made to the properties of the button and form. At times, I'll show you the structure of the form of the examples as it appears once it has been converted in a readable format (I'll describe how to convert a form into text later in this chapter). I won't show you the entire textual description of a form (which is often quite long), but rather only its key elements. I won't include the lines describing the position of the components, their sizes, or some less important default values. Here is the code:

```
object Form1: TForm1
  Caption = 'Hello'
  OnClick = FormClick
  object BtnHello: TButton
    Caption = 'Say hello'
    OnClick = BtnHelloClick
  end
```

```
end
```

This description shows some attributes of the components and the events they respond to. We will see the code for these events in the following sections. If you run this program now, you will see that the button works properly. In fact, if you click on it, it will be pushed, and when you release the mouse button, the on-screen button will be released. The only problem is that when you press the button, you might expect something to happen; but nothing does, because we haven't assigned any action to the mouse-click yet.

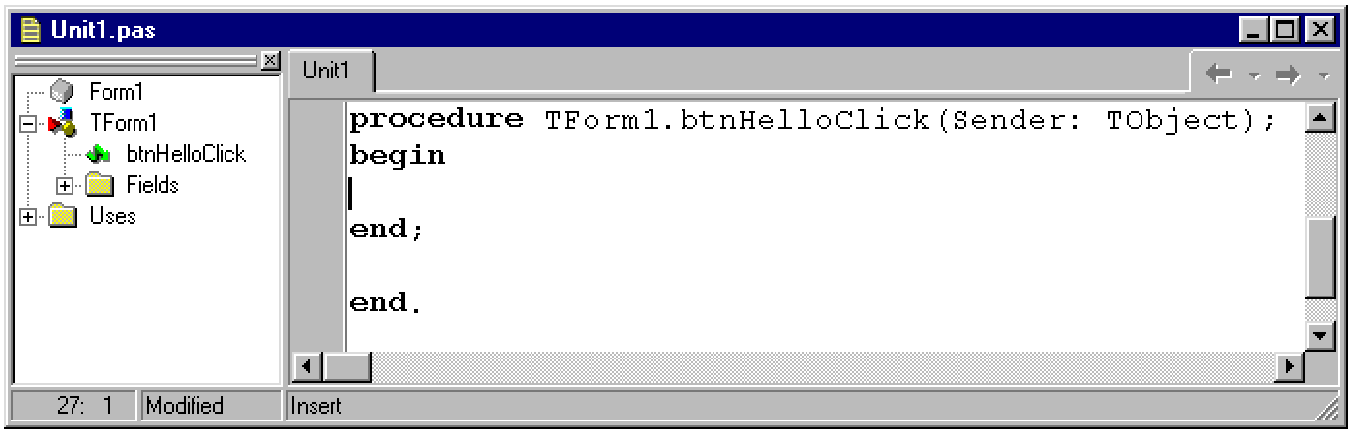
Responding to Events

When you press the mouse button on a form or a component, Windows informs your application of the event by sending it a message. Delphi responds by receiving an event notification and calling the appropriate event-handler method. As a programmer, you can provide several of these methods, both for the form itself and for the components you have placed in it. Delphi defines a number of events for each kind of component. The list of events for a form is different from the list for a button, as you can easily see by clicking on these two components while the Events page is selected in the Object Inspector. Some events are common to both components.

There are several techniques you can use to define a handler for the `OnClick` event of the button:

- Select the button, either in the form or by using the Object Inspector's combo box (called the Object Selector), select the Events page, and double-click in the white area on the right side of the OnClick event. A new method name will appear, *BtnHelloClick*.
- Select the button, select the Events page, and enter the name of a new method in the white area on the right side of the `OnClick` event. Then press the Enter key to accept it.
- Double-click on the button, and Delphi will perform the default action for this component, which is to add a handler for the `OnClick` event. Other components have completely different default actions.

With any of these approaches, Delphi creates a procedure named `BtnHelloClick` (or the name you've provided) in the code of the form and opens the source code file in that position:



```
procedure TForm1.btnHelloClick(Sender: TObject);
begin

end;

end.
```

The default action for a button is to add a procedure to respond to the click event. Even if you are not sure of the effect of the default action of a component, you can still double-click on it. If you end up adding a new procedure you don't need, just leave it empty. Empty method bodies generated by Delphi will be removed as soon as you save the file. In other words, if you don't put any code in them, they simply go away.

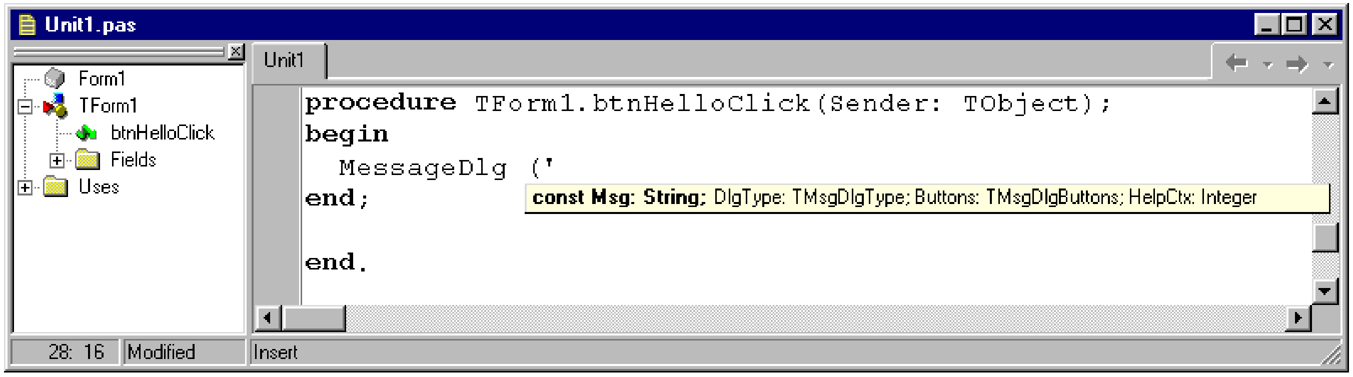
When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event-handler that is still empty, consider adding a comment to it, so that it will not be removed.

Now we can start typing some instructions between the `begin` and `end` keywords that delimit the code of the procedure. Writing code is usually so simple that you don't need to be an expert in the language to start working with Delphi. (If you need to brush up your knowledge of Pascal you can refer to my online Essential Pascal book, while if you need derailede coverage of Object Pascal you can refer to my Mastering Delphi series.)

You should type only the line in the middle, but I've included the whole source code of the procedure to let you know where you need to add the new code in the editor:

```
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
end;
```

The code is simple. There is only a call to a function, `MessageDlg`, to display a small message dialog box. The function has four parameters. Notice that as you type the open parenthesis, the Delphi editor will show you the list of parameters in a hint window, making it simpler to remember them.



```
procedure TForm1.btnHelloClick(Sender: TObject);
begin
  MessageDlg ('
end;

end.
```

const Msg: String; DlgType: TMsgDlgType; Buttons: TMsgDlgButtons; HelpCtx: Integer

If you need more information about the parameters of this function and their meanings, you can click on its name in the edit window and press F1. This brings up the Help information. Since this is the first code we are writing, here is a summary of that description (the rest of this book, however, generally does *not* duplicate the reference information available in Delphi's Help system, concentrating instead on examples that demonstrate the features of the language and environment):

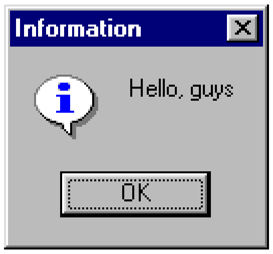
- The first parameter of the MessageDlg function is the string you want to display: the message.

- The second parameter is the type of message box. You can choose mtWarning, mtError, mtInformation, or mtConfirmation. For each type of message, the corresponding caption is used and a proper icon is displayed at the side of the text.
- The third parameter is a set of values indicating the buttons you want to use. You can choose mbYes, mbNo, mbOK, mbCancel, or mbHelp. Since this is a set of values, you can have more than one of these values. Always use the proper set notation with square brackets ([and]) to denote the set, even if you have only one value, as in the line of the code above. (Essential Pascal discusses Pascal sets.)
- The fourth parameter is the help context, a number indicating which page of the Help system should be invoked if the user presses F1. Simply write `0` if the application has no help file, as in this case.

The function also has a return value, which I've just ignored, using it as if it were a procedure. In any case, it's important to know that the function returns an identifier of the button that the user clicked to close the message box. This is useful only if the message box has more than one button.

Programmers unfamiliar with the Pascal language, particularly those who use C/C++, might be confused by the distinction between a function and a procedure. In Pascal, there are two different keywords to define procedures and functions. The only difference between the two is that functions have a return value.

After you have written this line of code, you should be able to run the program. When you click on the button, you'll see the message box shown below.



Every time the user clicks on the push button in the form, a message is displayed. What if the mouse is pressed outside that area? Nothing happens. Of course, we can add some new code to handle this event. We only need to add an `OnClick` event to the form itself. To do this, move to the Events page of the Object Inspector and select the form. Then double-click at the right side of the `OnClick` event, and you'll end up in the proper position in the edit window. Now add a new call to the `MessageDlg` function, as in the following code:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;
```

With this new version of the program, if the user clicks on the button, the hello message is displayed, but if the user misses the button, a warning message appears. Notice that I've written the code on two lines, instead of one. The Pascal compiler completely ignores new lines, white spaces, tab spaces, and similar formatting characters. Program statements are separated by semicolons (`;`), not by new lines.

There is one case in which Delphi doesn't completely ignore line breaks: Strings cannot extend across multiple lines. In some cases, you can split a very long string into two different strings, written on two lines, and merge them by writing one after the other.

Compiling and Running a Program

Before we make any further changes to our Hello program, let's stop for a moment to consider what happens when you run the application. When you click on the toolbar Run button or select Run | Run, Delphi does the following:

1: Compiles the Pascal source code file describing the form.

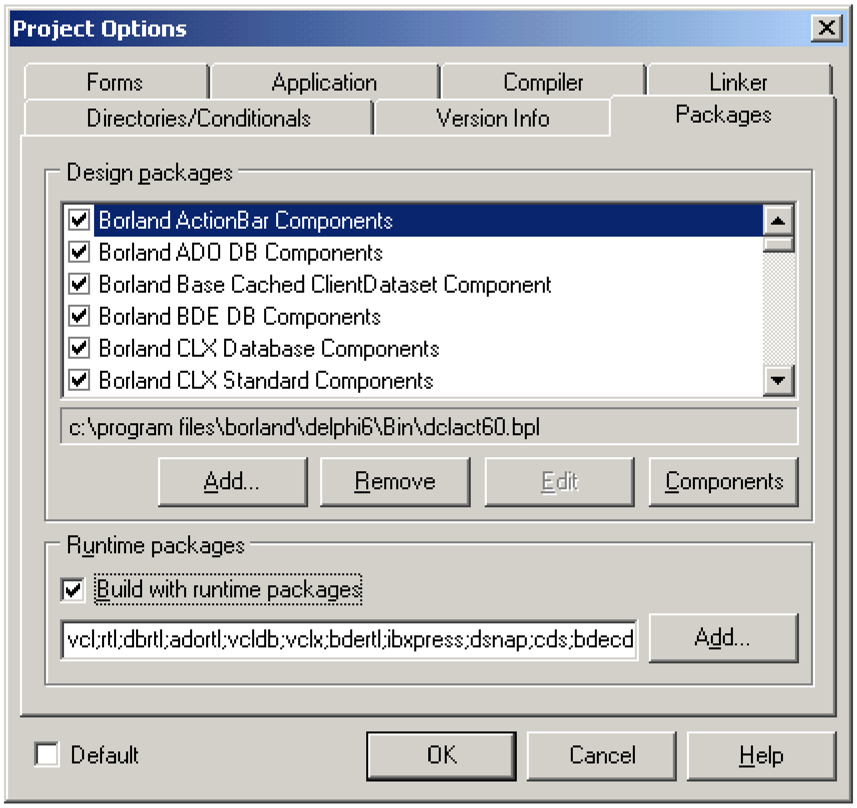
2: Compiles the project file.

3: Builds the executable (`EXE`) file, linking the proper libraries.

4: Runs the executable file, usually in debug mode.

In early versions of Delphi, the executable file you obtained was invariably a stand-alone program. Starting with version 3, Delphi allows you to link all the required libraries into the executable file, but you can also specify the use of separate run-time packages, making the executable file much smaller.

The key point is that when you ask Delphi to run your application, it compiles it into an executable file. You can easily run this file from the Windows Explorer or using the Run command on the Start button. Compiling this program as usual, linking all the required library code, produces an executable of about a couple of hundred Kb. By using run-time packages, this can shrink the executable to about 20 Kb. Simply select the Project | Options menu command, move to the Packages page, and select the check box *Build with runtime packages*:



Design packages:
- ✓ Borland ActionBar Components
- ✓ Borland ADO DB Components
- ✓ Borland Base Cached ClientDataset Component
- ✓ Borland BDE DB Components
- ✓ Borland CLX Database Components
- ✓ Borland CLX Standard Components

c:\program files\borland\delphi6\Bin\dclact60.bpl

✓ Build with runtime packages

vcl;rtl;dbrtl;adortl;vcldb;vclx;bdertl;ibxpress;dsnap;cds;bdecd

Packages are dynamic link libraries containing Delphi components (the Visual Components Library). By using packages you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as `vcl60.bpl`) are available on the computer where you want to run the program. The `BPL` extensions stands for Borland Package Libraries; it is the extension used by Delphi (and C++Builder) packages, which are technically DLL files. Using this extension makes it easier to recognize them (and find them on a hard disk).

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 program built with run-time packages is much bigger than the space required by the bigger stand-alone executable file. For this reason the use of packages is not always recommended. The great advantage of Delphi over competing development tools is that you can easily choose whether to use the stand-alone executable or the small executable with run-time packages.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable with that of a C or C++ program. Delphi compiled code runs much faster (at least 10 times faster) than the equivalent code in interpreted or *semi-compiled* tools.

Some users cannot believe that Delphi generates real executable code, because when you run a small program, its main window appears almost immediately, as happens in some interpreted environments. To see for yourself, try this: Open the Environment Options dialog box (using Tools | Options), move to the Preferences page, and turn on the Show Compile Progress option. Now select Project | Build All. You'll see a dialog box with the compilation status. You'll find that this takes just a few seconds, or even less on a fast machine.

In the tradition of Borland's Turbo Pascal compilers, the Object Pascal compiler embedded in Delphi works very quickly. For a number of technical reasons, it is much faster than any C++ compiler. If you try using the new Borland C++ Builder development environment (which is very similar to Delphi) the compilation requires more time, particularly the first time you build an application. One reason for the higher speed of the Delphi compiler is that the language definition is simpler. Another is that the Pascal compilers and linkers have less work to do to include libraries or other compiled source files in a program, because of the structure of units.

Changing Properties at Run-Time

Let's return to the Hello application. We now want to try to change some properties at run-time. For example, we might change the text of `HelloButton` from *Say hello* to *Say hello again* after the first time a user clicks on it. You may also need to widen the button, as the caption becomes longer. This is really simple. You only need to change the code of the `HelloButtonClick` procedure as follows:

```
procedure TForm1.HelloButtonClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  btnHello.Caption := 'Say Hello Again';
end;
```

The Pascal language uses the `:=` operator to express an assignment and the `=` operator to test for equality. At the beginning, this can be confusing for programmers coming from other languages. For example in C and C++, the assignment operator is `=`, and the equality test is `==`. After a while, you'll get used to it. In the meantime, if you happen to use `=` instead of `:=`, you'll get an error message from the compiler.

A property such as `Caption` can be changed at run-time very easily, by using an assignment statement. Most properties can be changed at run-time, and some can be changed *only* at run-time. You can easily spot this last group: They are not listed in the Object Inspector, but they appear in the Help file for the component. Some of these run-time properties are defined as read-only, which means that you can access their value but cannot change it.

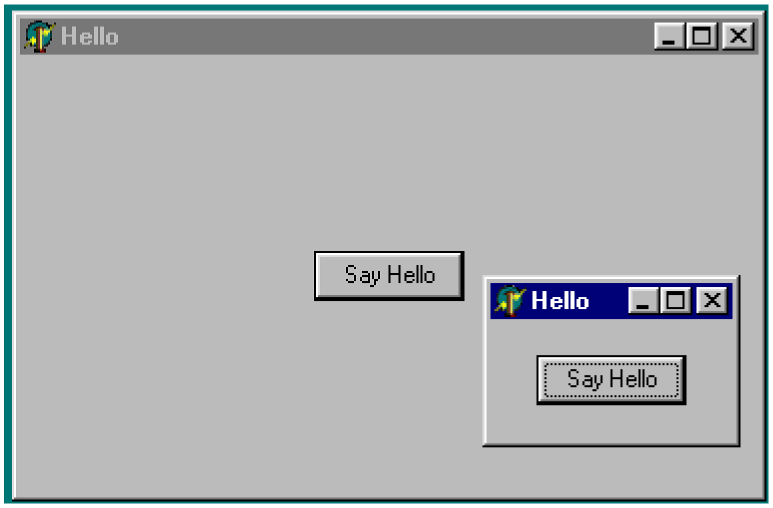
Adding Code to the Program

Our program is almost finished, but we still have a problem to solve, which will require some real coding. The button starts in the center of the form, but will not remain there when you resize the form. This problem can be solved in two radically different ways.

One solution is to change the border of the form to a thin frame, so that the form cannot be resized at run-time. Just move to the `BorderStyle` property of the form, and choose `bsSingle` instead of `bsSizeable` from the combo box. The other approach is to write some code to move the button to the center of the form each time the form is resized, and that's what we'll do next. Although it might seem that most of your work in programming with Delphi is just a matter of selecting options and visual elements, there comes a time when you need to write code. As you become more expert, the percentage of the time spent writing code will generally increase.

When you want to add some code to a program, the first question you need to ask yourself is Where? In an event-driven environment, the code is always executed in response to an event. When a form is resized, an event takes place: `OnResize`. Select the form in the Object Inspector and double-click next to `OnResize` in the Events page. A new procedure is added to the source file of the form. Now you need to type some code in the editor, as follows:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;
```



To set the `Top` and `Left` properties of the button — that is, the position of its upper-left corner — the program computes the center of the frame, dividing the height and the width of the internal area or client area of the frame by 2, and then subtracts half the height or width of the button. Note also that if you use the `Height` and `Width` properties of the form, instead of the `ClientWidth` and `ClientHeight` properties, you will refer to the center of the whole window, including the caption at the top border. This final version of the example works quite well as you can see below.

This figure includes two versions of the form, with different sizes. By the way, this figure is a real snapshot of the screen. Once you have created a Windows application, you can run several copies of it at the same time by using the Explorer. By contrast, the Delphi environment can run only one copy of a program. When you run a program within Delphi, you start the integrated debugger, and it cannot debug two programs at the same time — not even two copies of the same program — unless you are using Windows NT/2000/XP.

A Two-Way Tool

In the Hello example, we have written three small portions of code, to respond to three different events. Each portion of code was part of a different procedure (actually a method, as you'll learn reading Chapter 5). But where does the code we write end up? The source code of a form is written in a single Pascal language source file, the one we've named HELLOF.PAS. This file evolves and grows not only when you code the response of some events, but also as you add components to the form. The properties of these components are stored together with the properties of the form in a second file, named HELLOF.DFM.

Delphi can be defined as a two-way tool, since everything you do in the visual environment ends up in some code. Nothing is hidden away and inaccessible. You have the complete code, and although some of it might be fairly complex, you can edit everything. Of course, it is easier to use only the visual tools, at least until you are an expert Delphi programmer.

The term *two-way tool* also means that you are free to change the code that has been produced, and then go back to the visual tools. This is true as long as you follow some simple rules.

Looking at the Source Code

Let's take a look at what Delphi has generated from our operations so far. Every action has an effect — in the Pascal code, in the code of the form, or in both. When you start a new, blank project, the empty form has some code associated with it, as in the following listing.

```
unit Unit1;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

The file, named Unit1, uses a number of units and defines a new data type (a class) and a new variable (an object of that class). The class is named TForm1, and it is derived from TForm. The object is Form1, of the new type TForm1.

> *Units* are the modules into which a Pascal program is divided. When you start a new project, Delphi generates a program module and a unit that defines the main form. Each time you add a form to a Delphi program, you add a new unit. Units are then compiled separately and linked into the main program. By default, unit files have a `.PAS` extension and program files have a `.DPR` extension.

If you rename the files as suggested in the example, the code changes slightly, since the name of the unit must reflect the name of the file. If you name the file `Hellof.pas`, the code begins with

```
unit Hellof;
```

As soon as you start adding new components, the form class declaration in the source code changes. For example, when you add a button to the form, the portion of the source code defining the new data type becomes the following:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    ...
```

Now if you change the button's `Name` property (using the Object Inspector) to `BtnHello`, the code changes slightly again:

```
type
  TForm1 = class(TForm)
    BtnHello: TButton;
    ...
```

Setting properties other than the name has no effect in the source code. The properties of the form and its components are stored in a separate form description file (with a `DFM` extension).

Adding new event handlers has the biggest impact on the code. Each time you define a new handler for an event, a line is added to the data type definition of the form, an empty method body is added in the implementation part, and some information is stored in the form description file, too.

```
unit HelloForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    btnHello: TButton;
    procedure btnHelloClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.btnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  btnHello.Caption := 'Say Hello Again';
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;

end.
```

It is worth noting that there is a single file for the whole code of the form, not just small fragments. Of course, the code is only a partial description of the form. The source code determines how the form and its components react to events. The form description (the `DFM` file) stores the values of the properties of the form and of its components. In general, source code defines the actions of the system, and form files define the initial state of the system.

The Textual Description of the Form

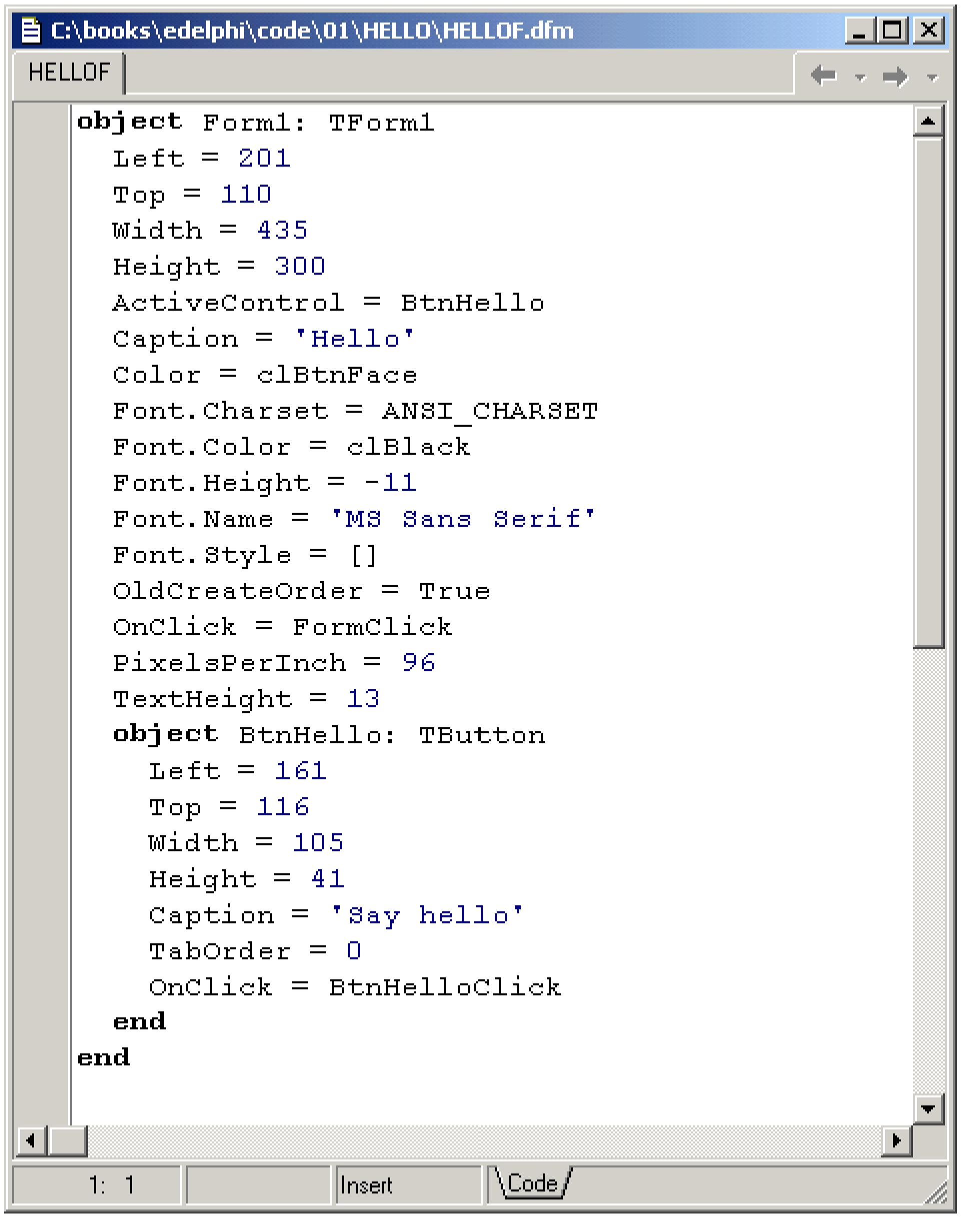
As I've just mentioned, along with the `PAS` file containing the source code, there is another file describing the form, its properties, its components, and the properties of the components. This is the `DFM` file, a binary or text file (this latter option has been introduced with Delphi 5). Whatever the format, if you load this file in the Delphi code editor, it will be converted into a textual description. This might give the false impression that the `DFM` file is indeed a text file, but this can be only if you've selected the corresponding option (available since Delphi 5).

You can open the textual description of a form simply by selecting the shortcut menu of the form designer (that is, right-clicking on the surface of the form at design-time) and selecting the View as Text command. This 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 of the local menu of the editor window. The alternative is to open the `DFM` file directly in the Delphi editor.

To understand what is stored in the DFM file, you can look at the next listing, which shows the textual description of the form of the first version of the Hello example. This is exactly the code you'll see if you give the View as Text command in the local menu of the form:

```
object Form1: TForm1
  Left = 235
  Top = 108
  Width = 430
  Height = 308
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  OnCreate = FormCreate
  OnResize = FormResize
  PixelsPerInch = 96
  TextHeight = 13
  object btnHello: TButton
    Left = 165
    Top = 111
    Width = 75
    Height = 25
    Caption = 'Say Hello'
    TabOrder = 0
    OnClick = btnHelloClick
  end
end
```

You can compare this code with what I used before to indicate the key features and properties of the form and its components. As you can see in this listing, the textual description of a form contains a number of objects (in this case, two) at different levels. The Form1 object contains the BtnHello object, as you can immediately see from the indentation of the text. Each object has a number of properties, and some methods connected to events (in this case, OnClick).



```
object Form1: TForm1
  Left = 201
  Top = 110
  Width = 435
  Height = 300
  ActiveControl = BtnHello
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = ANSI_CHARSET
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  OnClick = FormClick
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 161
    Top = 116
    Width = 105
    Height = 41
    Caption = 'Say hello'
    TabOrder = 0
    OnClick = BtnHelloClick
  end
end
```

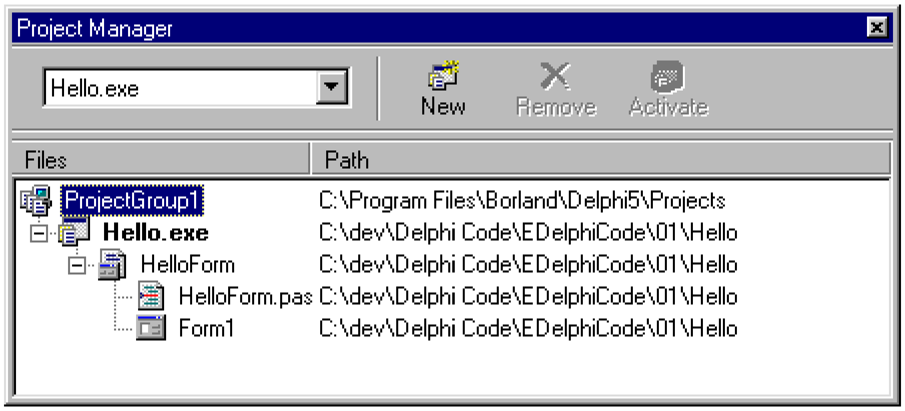
Once you've opened this file in Delphi, you can *edit* the textual description of the form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, this compilation will stop with an error message, and you'll need to correct the contents of your `DFM` file before you can reopen the form in the editor. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

An expert programmer might choose to work on the text of a form for a number of reasons. For big projects, the textual description of the form is a powerful documenting tool, an important form of backup (in case someone plays with the form, you can understand what has gone wrong by comparing the two textual versions), and a good target for a version-control tool. For these reasons, Delphi also provides a DOS command-line tool, `CONVERT.EXE`, which can translate forms from the compiled version to the textual description and vice versa.

As we will see in the next chapter, the conversion is also applied when you cut or copy components from a form to the Clipboard.

The Project File

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR). This file is built automatically, and you seldom need to change it, particularly for small programs. If you do need to change the behavior of a project, there are basically two ways to do so: You can use the Delphi Project Manager and set some project options, or you can manually edit the project file directly.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi5\Projects
Hello.exe	C:\dev\Delphi Code\EDelphiCode\01\Hello
HelloForm	C:\dev\Delphi Code\EDelphiCode\01\Hello
HelloForm.pas	C:\dev\Delphi Code\EDelphiCode\01\Hello
Form1	C:\dev\Delphi Code\EDelphiCode\01\Hello

This project file is really a Pascal language source file, describing the overall structure of the program and its startup code:

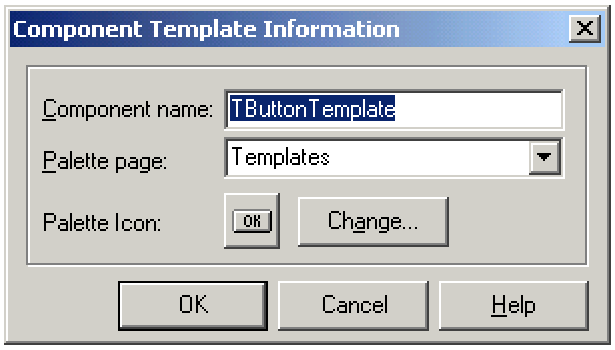
```
program Hello;

uses
  Forms,
  Hellof in 'HelloForm.PAS' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

You can see this file with the View | Project Source menu command. As an alternative, you can click on the Select unit button of the toolbar or issue the equivalent menu command, View | Units. When you use one of these commands, Delphi shows a dialog box with the list of the source files of the project. You can choose the project file (named Hello in the example), or any other file you are interested in seeing.

Using Component Templates

Suppose you want to create a brand new application, with a similar button and a similar event handler to the Hello program. It is possible to copy a component to the Clipboard, and then paste it into another form to create a perfect clone. However, doing so you copy only the properties of the component, and not the events associated with it.

Delphi allows you to copy one or more components, and install them as a new component template. This way, you also copy the code of the methods connected with the events of the component. Simply open the Hello example, or any other one, select the component you want to move to the template (or a group of components), and then select the Component | Create Component Template menu command. This opens the Component Template Information dialog box, shown below. Here you enter the name of the template, the page of the Component Palette where it should appear, and an icon.


Component Template Information

Component name: TButtonTemplate

Palette page: Templates

What's Next

In this chapter, we created a simple program, added a button to it, and handled some basic events, such as a click with the mouse or a resize operation. We also saw how to name files, projects, forms, and components, and how this affects the source code of the program. We looked at the source code of the simple programs we've built, although some of you might not be fluent enough in Object Pascal to understand the details.

Before we can look into more complex examples, we need to explore the Delphi development environment. This is the topics of the next chapter. The examples in this chapter should have shown you that Delphi is really easy to use. Now we'll start to look at the complex mechanisms behind the scenes that make this all possible. You'll see that Delphi is a very powerful tool, even though you can use it to write programs easily and quickly.

Chapter 2: Highlights of the Delphi Environment

In a visual programming tool such as Delphi, the role of the environment is certainly important, at times even more important than the programming language used by its compiler or interpreter. This is a good reason to spend some time reading this chapter.

This chapter won't discuss all of the features of Delphi or list all of its menu commands. Instead, it will give you the overall picture and help you to explore some of the environment traits that are not obvious, while suggesting some tips that may help you. You'll find more information about specific commands and operations throughout the book.

Different Versions 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 version of Delphi; there are three of them:

- The basic version (the "Standard" edition) is aimed at Delphi newcomers and casual programmers. It has all the features required to write programs in Delphi for Windows but (starting with Delphi 5) it has no support for any type of database programming.
- The second level (the "Professional" edition) is aimed at professional developers. It includes database support (with BDE and ODBC connectivity), limited Web support, and many more components.
- The full-blown Delphi (the "Enterprise" edition starting with Delphi 5, or the "Client/Server Suite" edition in previous versions) is aimed at developers building client/server applications. It includes also drivers for native Client/Server connection, full ADO support, MIDAS and Internet Express support, and (in Delphi) all of the new Snap technologies: BizSnap, WebSnap, and DataSnap.

Besides the different editions available, there are a number of ways to customize the Delphi environment. You can change the buttons of the toolbar, attach new commands to the Tools menu, hide some of the windows or elements, and resize and move all of them. In the screen illustrations throughout the book, I'll try to use a standard user interface (as it comes out of the box); however, I have my preferences, and I generally install many add-ons (written by third parties or by me), which might be reflected in some of the screen shots.

Asking for Help

Now we can really start our tour. The first element of the environment we'll explore is the Help system. There are basically two ways to invoke the Help system: select the proper command in the Help pull-down menu, or choose an element of the Delphi interface or a token in the source code and press F1.

> When you press F1, Delphi doesn't search for an exact match in the Help Search list. Instead, it tries to understand what you are asking. For example, if you press F1 when the text cursor is on the name of the Button1 component in the source code, the Delphi Help system automatically opens the description of the TButton class, since this is what you are probably looking for. This technique also works when you give the component a new name. Try naming the button Foo, then move the cursor to this word, press F1, and you'll still get the help for the TButton class. This means Delphi looks at the contextual meaning of the word for which you are asking help.

Note that there isn't just a single help file in Delphi. Most of the time, you'll invoke Delphi Help, but this file is complemented by an Object Pascal Help file, the Windows API Help, the Component Writer's Help, and many others (depending on your version of Delphi). These and other Help files have a common outline and a common search engine you can activate by pressing the Help Topics button while in the Help system. The Windows help engine dialog box that appears allows you to browse the contents of all of the help files in the group, search for a keyword in the index, or start the Find engine. The three capabilities are available in separate pages of the Help Topics dialog box.

You can find almost everything in the Help system, but you need to know what to search for. Usually this is obvious, but at times it is not. Spending some time just playing with the Help system will probably help you understand the structure of these files and learn how to find the information you need.

The Help files provide a lot of information, both for beginner and expert programmers, and they are especially valuable as a reference tool. They list all of the methods and properties for each component, the parameters of each method or function, and similar details, which are particularly important while you are writing code. Borland also distributes reference materials in the form of Adobe Acrobat files. These are electronic versions of the printed manuals that come in the Delphi box, so you can search them for a word, and you can also print the portions you are interested in (or even the whole file if you've got some spare paper).

> The first version of Delphi included some Interactive Tutors in addition to the Help system. If you've never used Delphi (and if you have Delphi 1 installed), you might consider running these Tutors. They will guide you through Delphi's basic features and help you understand some of the terminology of the environment. Unluckily they were soon discontinued by Borland.

Besides the Delphi Help files, there are many sources of collections of tips and suggestions for Delphi programmers. Borland web sites provides some FAQs (Frequently Asked Questions) and a collection of Technical Information short papers (TI). You can find updates of both at the Borland Community Web site (http://community.borland.com). Besides these official Borland documents, you'll find many more tips in Borland newsgroups (and also on mine). Obviously the Web is a great source of information about Delphi itself and third party products. You can find a collection of my favorite Delphi Web pages in the Links portion of my web site (http://www.marcocantu.com/links).

Delphi Menus and Commands

There are basically three ways to issue a command in the Delphi environment:

- Use the menu.

- Use the toolbar.
- Use one of the local menus activated by pressing the right mouse button.

The Delphi menus offer many commands. I won't bore you with a detailed description of the menu structure. For this type of information, you can refer to the printed documentation or the Help file. In the following sections, I'll present some suggestions on the use of some of the menu commands. Other suggestions will follow in the rest of the chapter.

The File Menu

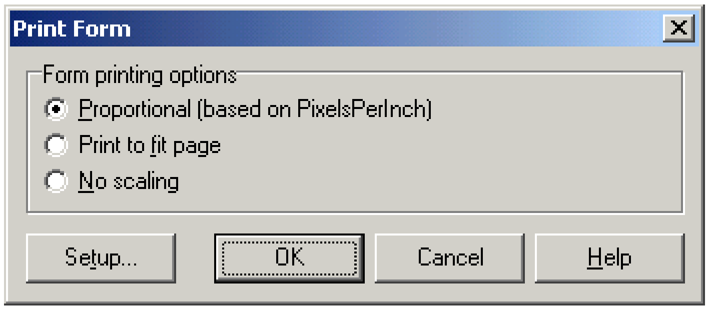
Our starting point is the File pull-down menu. The structure of this menu has kept changing from version to version of Delphi, with menu items for handling projects moving away, and specific commands to create new designer (for example data modules) coming and going. Still, this menu contains commands that operate on projects and commands that operate on source code files.

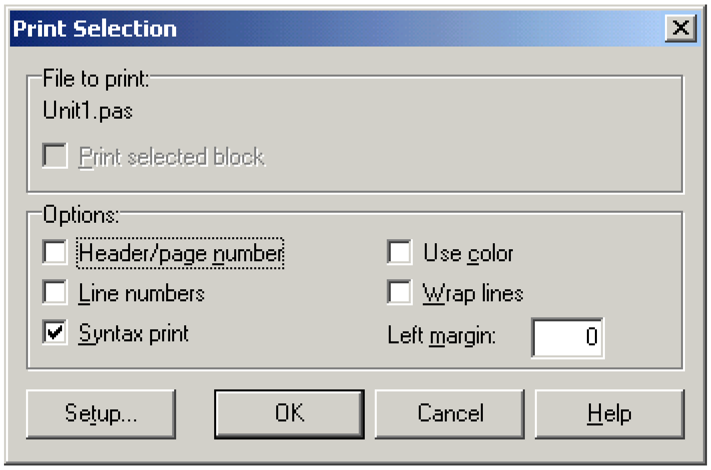
*[*** The File menu structure has changed in Delphi 6, with the File | New submenu, and the text here has not been updated accordingly]* Some commands can even be used to operate both on projects and on source code files. The commands related to projects are New, New Application, Open, Save Project As, Save All, Close All, Add to Project, and Remove from Project. Besides these, there is also a specific Project pull-down menu. The commands related to source code files are New, New Form, New Data Module, Open, Reopen, Save, Save As, Close, and Print. Most of these commands are very intuitive, but some require a little explanation.

Use the Reopen menu command to open projects or source code files you have worked on recently.

The New command actually opens the New Items dialog box, also called the Object Repository. This dialog box can be used to invoke Delphi Wizards and to create items such as new applications, forms that inherit from existing forms, threads, DLLs, Delphi components, and ActiveX controls. I'll cover the Object Repository's rich set of features in the next chapter.

Another peculiar command is Print. If you are editing source code and select this command, the printer will output the text with syntax highlighting as an option. If you are working on a form and select Print from the File menu, the printer will produce the graphical representation of the form. This is certainly nice, but it can be confusing, particularly if you are working on other Delphi windows. Fortunately, two different print options dialog boxes are displayed, so that you can check that the operation is correct.





File to print: Unit1.pas
Print selected block: [ ]
Options:
Header/page number: [ ]
Line numbers: [ ]
Syntax print: [✓]
Use color: [ ]
Wrap lines: [ ]
Left margin: 0

The Edit Menu

The Edit menu has some typical operations, such as Undo and Redo, and the Cut, Copy, and Paste commands, plus some specific commands for form or editor windows. The important thing to notice is that the standard features of the Edit menu (and the standard Ctrl+Z, Ctrl+X, Ctrl+C, and Ctrl+V keyboard shortcuts) work both with text and with form components. There are also some differences worth noting. For example, when you work with the editor, the first command of this pull-down menu is Undo; when you work with the form, it becomes Undelete. Unfortunately, the Form Designer has very limited Undo capabilities.

Of course, you can copy and paste some text in the editor, and you can also copy and paste components in one form, or from one form to another. You can even paste components to a different parent window of the same form, such as a panel or group box.

Besides using cut and paste commands, the Delphi editor allows you to move source code by selecting and dragging words, expressions, or lines. If you drag text while pressing the Ctrl key, it will be copied instead of moved.

Copying and Pasting Components

What you might not have noticed is that you can also copy components from the form to the editor and vice versa. Delphi places components in the Clipboard along with their textual description. 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 = 56
  Top = 48
  Width = 161
  Height = 57
  TabOrder = 0
  Caption = 'Button1'
end
```

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

```
object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  TabOrder = 0
  Caption = 'My Button'
  Font.Name = 'Arial'
end
```

Copying the above 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 make use of 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 change the position of the component so that it doesn't overlap a previous copy. You can see how Delphi modifies the properties of the component by copying it back to the editor. For example, this is what you get if you paste the text above in the form, and then copy it again into the editor:

```
object MyButton: TButton
  Left = 112
  Top = 128
  Width = 180
  Height = 60
  Caption = 'My Button'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
end
```

As you can see, some lines have been added automatically, to specify other properties of the font. Of course, if you write something completely wrong, such as this code:

```
object Button3: TButton
  Left = 100
  eight = 60
end
```

which has a spelling error (a missing 'H'), and try to paste it into a form, Delphi will show an error 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 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.

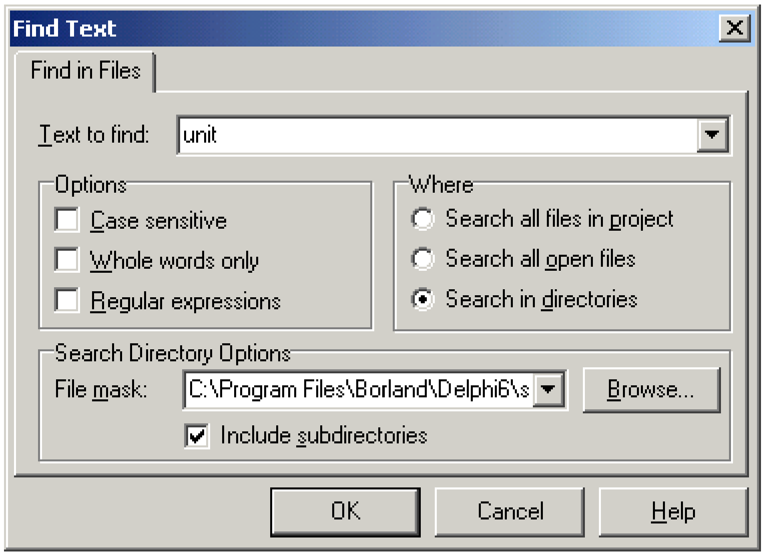
More Edit Commands

Along with the typical commands found on most Edit menus in Windows applications, Delphi includes a number of commands that are mostly related to forms. The specific operations for forms can also be accessed through the form shortcut menu (the local menu you can invoke with the right mouse button) and will be covered later in the chapter.

One command not replicated in a form's local menu is Lock Controls, which is very useful for avoiding an accidental change to the position of a component in a form. For example, you might try to double-click on a component and actually end up moving it. Since there is no Undo operation on forms, protecting from similar errors by locking the controls after the form has been designed can be really useful.

The Search Menu

The Search menu has some standard commands, too, such as Search and Replace, and the Find in Files command (you can see its dialog box here):


Find Text

Find in Files

Text to find: unit

Options
- [ ] Case sensitive
- [ ] Whole words only
- [ ] Regular expressions

Where
- ○ Search all files in project
- ○ Search all open files
- ● Search in directories

Search Directory Options

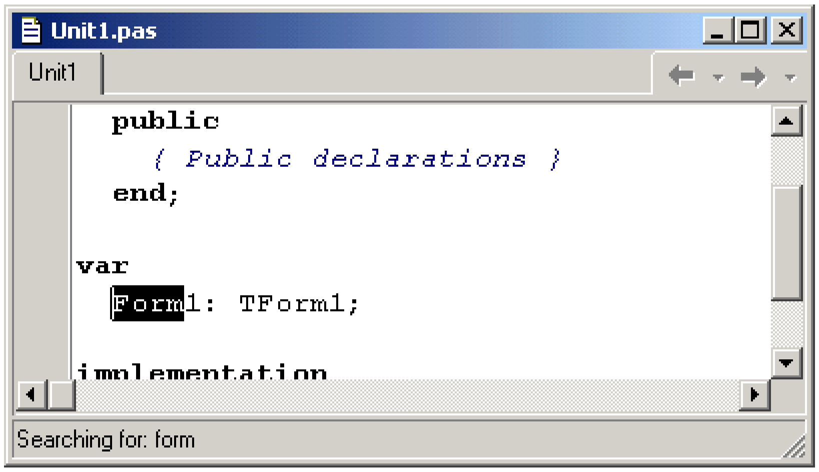
File mask: C:\Program Files\Borland\Delphi6\s

- [✓] Include subdirectories

The Find in Files command allows you to search for a string in all of the source code files of a project, all the open files, or all the files in a directory (optionally including its subdirectories), depending on the radio button you check. The result of the search will be displayed in the message area at the bottom of the editor window. You can select an entry to open the corresponding file and jump to the line containing the text.

You can use the Find in Files command to search for component, class, and type definitions in the VCL source code (if your version of Delphi includes it). This is an easy way to get detailed information on a component, although using Help is generally faster and simpler.

Other commands are not so simple to understand. The Incremental Search command is one of them. When you select this command, instead of showing a dialog box where you enter the text you want to find, Delphi moves to the editor. There, you can type the text you want to search for directly in the editor message area, as you can see here:



```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

Searching for: form

When you type the first letter, the editor will move to the first word starting with that letter. (But if your search text isn't found, the letters you typed won't even be displayed in the editor message area.) If that is not the word you are looking for, just keep typing; the cursor will continue to jump as you add letters. Although this command might look strange at first, it is very effective and extremely fast, particularly if you are typing and invoke it with a shortcut key (Ctrl+E if you are using the standard editor shortcuts).

The Go to Line Number command is quite intuitive. The Find Error command might seem strange at first. It is used to find a particular run-time error, not to search for a compiler error. When you are running a stand-alone program and you hit a very bad error, Delphi displays an internal address number (that is, the *logical* address of the compiled code). You can enter this value in the Find Error dialog box to have Delphi recompile the program, looking for the specific address. When it finds the address, Delphi shows the corresponding source code line. Often, however, the error is not in one of the lines of your code, but in a line of library or system code; in this (quite frequent) case the Find Error command cannot locate the offending line.

The last command on the Search menu, Browse Symbol, invokes the Object Browser, a tool you can use to explore all the symbols defined in a compiled program. To understand the output of the Object Browser, you need a good understanding of the Object Pascal language and of the Visual Component Library (VCL).

The View Menu

The View pull-down menu combines the features you usually find in View and Window menus. There is no Window menu, because the Delphi environment is not an MDI application. Most of the View commands can be used to display one of the windows of the Delphi environment, such as Project Manager, the Breakpoints list, or the Components command. Some of these windows are used during debugging; others when you are writing code. Most of these windows will be described later in this chapter.

It is possible to add a new item to the View menu, CPU Window, which can be used during debugging to view the Assembler code generated by the Delphi compiler, execute it step by step, and view the status of the CPU registers.

The commands on the second part of the View menu are important, which is why they are also available on the default toolbar. The Toggle Form/Unit (or F12) command is used to move between the form you are working on and its source code. If you use a source code window big enough to hold a reasonable amount of text, you'll use this command often. As an alternative, you can place the two windows (the editor and the form) so that a portion of the one below is always visible. With this arrangement, you can click on it with the mouse to move it to the front.

The New Edit Window command opens a second edit window. It is the only way to view two files side by side in Delphi, since the editor uses tabs to show the multiple files you can load. Once you have duplicated the edit window, you can make each one hold a different set of files, or view two portions of the same file.

The last two commands on the View menu can be used to hide the toolbar or the Components palette, although this is a good way to make Delphi look silly and uncomfortable. Working on forms without the Components palette is certainly not easy. If you remove both the toolbar and the Components palette, the Delphi main window is reduced to a bare menu.

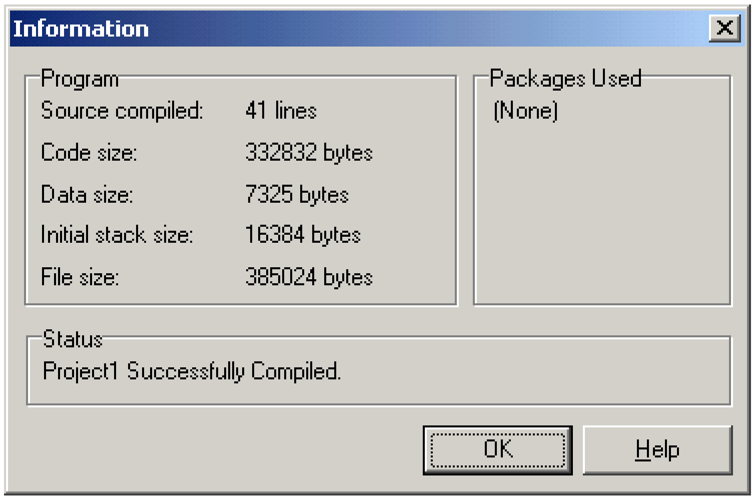
The Project Menu

The next pull-down menu, Project, has commands to manage a project and compile it. Add to Project and Remove from Project are used to add forms or Pascal source code files to a program and to remove them from a project.

The Enterprise version of Delphi includes two more commands, Web Deploy Options and Web Deploy, which are not available in the other editions. These features are related with ActiveX and ActiveForms, some (now obsolete) Microsoft web technologies.

The Compile command builds or updates the application executable file, checking which source files have changed and recompiling them when needed. With Build All, you can ask Delphi to compile every source file of the project, even if it has not been changed since the last compilation. If you just want to know whether the syntax of the code you've written is correct, but you do not want to build the program, you can use the Syntax Check command.

The next Project command, Information, displays some details about the last compilation you've made. The following figure shows the information related to the compilation of the a short program:



Program

Source compiled: 41 lines

Code size: 332832 bytes

Data size: 7325 bytes

Initial stack size: 16384 bytes

File size: 385024 bytes

Packages Used

(None)

Status

Project1 Successfully Compiled.

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 (Delphi Compiled Unit).

At the end of the Project menu comes the Options menu, used to set compiler and linker options, application object options, and so on. When you change the project options, you can check the Default box to indicate that the same set of options should be used for new projects. We will discuss project options again in this chapter and then throughout the book in the context of related topics.

The Run Menu

The Run menu could have been named Debug as well. Most of its commands are related to debugging, including the Run command itself. When you run a program within the Delphi environment, you execute it under the integrated debugger (unless you disable the corresponding Environment option). The Run command and the corresponding toolbar icon are among the most commonly used commands, since Delphi automatically recompiles a program before running it — at least if the source code has changed. Simply hit F9 as a shortcut to compile and run a program.

The next command, Parameters, can be used to specify parameters to be passed on the command line to the program you are going to run, and to provide the name of an executable file when you want to debug a DLL (DLL debugging is another new Delphi 3 feature). The remaining commands are all used during debugging, to execute the program step by step, set breakpoints, inspect the values of variables and objects, and so on. Some of these debugging commands are also available directly in the editor local menu.

Delphi 3 has a couple of new commands related to ActiveX development (and not found in the “Standard” edition). The Register ActiveX Server and Unregister ActiveX Server menu commands basically add or remove the Windows Registry information about the ActiveX control defined by the current project.

The Component Menu

The commands of the Component menu can be used to write components, add them to a package, or to install packages in Delphi. The New Component command invokes the simple Component Wizard. The three installation commands, Install Component, Import ActiveX Library, and Install Packages, can be used to add to the environment new Delphi components, packages, or ActiveX controls. Executing any of these commands adds the new components to the specified package and to the Components palette.

Component Templates

We briefly used the Create Component Template menu item in the last chapter. When you issue this command after selecting one or more components in a form, it opens a dialog box where you specify the name of the new component template, a page on the Palette, and an 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 Components palette (when Delphi displays the fly-by hint).

All the information about component templates is stored in a single file, DELPHI32.DCT, but there is apparently no 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 override the previous definition.

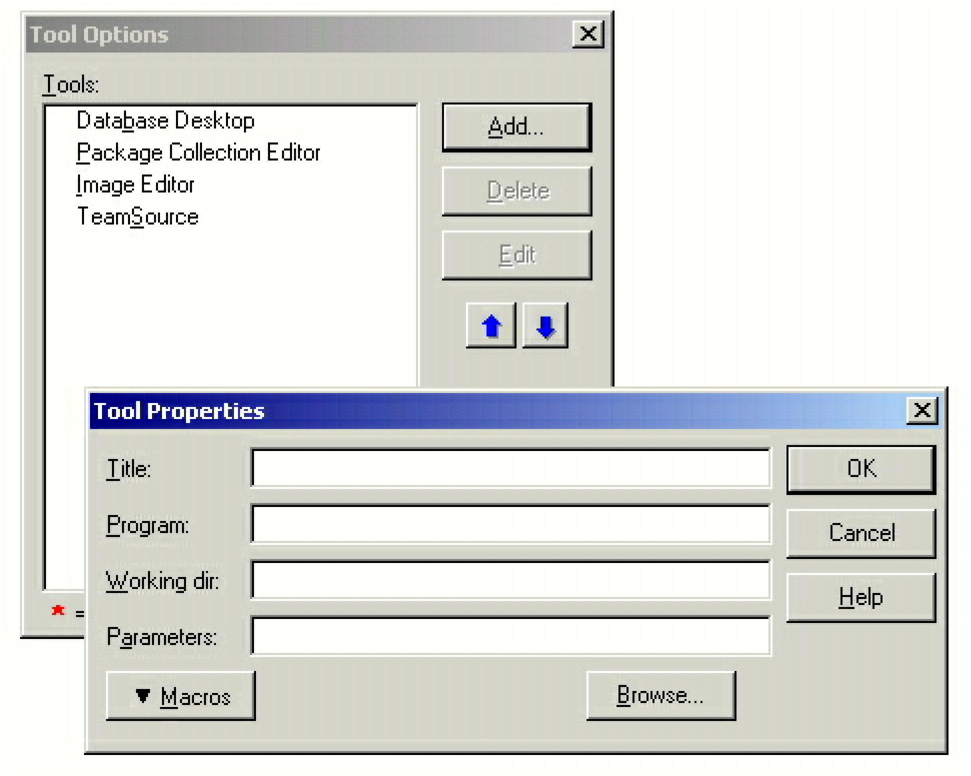
The Database Menu

The Database menu collects the Delphi database-related tools, such as the Database Form Wizard and the Database Explorer. The Enterprise edition has the SQL Explorer instead of the Database Explorer (although the menu items is invariably called Database | Explore) and a menu item to start the SQL Monitor.

The Tools Menu

The Tools menu simply lists a number of external programs and tools, just to make it easier to run them. You can use the Tools command to configure and add new external tools to the pull-down. Besides simply running a program, you can pass some parameters to it. Simple parameter lists can be passed directly on the command line, while complex ones can be built by clicking the Macros button in the lower part of the Tool Properties dialog box.

The Tools menu also includes a command to configure the Repository (discussed in the next chapter) and the Options command to configure the whole Delphi development environment. The Environment Options dialog box has many pages related to generic environment settings (the Preferences page), packages and library settings, many editor options (in the pages Editor, Display, and Colors), a page to configure the Components Palette, one for the object Browser, and one of the new Code Insight technology. I'll discuss many of these options when covering related features. You can see the dialog used to customize the Tools menu below.



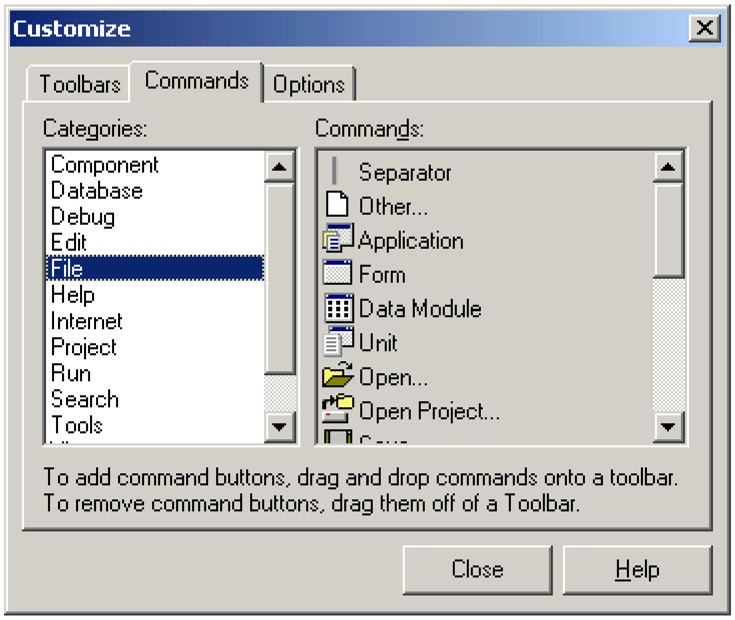
The Help Menu

The Help menu can be used to get information about Delphi (Help | Help Topics) and also to display the Delphi About box. In this window, you can hold down the Alt key and type the letters VERSION to see the Delphi version and build number. Using other key combinations (as mentioned in the acknowledgments at the beginning of the book) you can see a list of the people involved in building Delphi. The Help menu was often populated by third-party Delphi Wizards (before the new ToolsApi made it harder to place wizards in what was their default location).

The Delphi Toolbar

After you have used Delphi for a while, you'll realize that you use only a small subset of the available commands frequently. Some of these commands are probably already on the toolbar (Borland's name for a toolbar); some are not. If the commands you use a lot are not there, it's time to customize the toolbar so that it really helps you to use Delphi more efficiently.

An alternative to using the toolbar is to use shortcut keys. Although you must remember some key combinations to use them, shortcut keys let you invoke commands very quickly, particularly when you are writing code and your fingers are already on the keyboard.



To add command buttons, drag and drop commands onto a toolbar.
To remove command buttons, drag them off of a Toolbar.

You can easily resize the toolbar by dragging the thick line between it and the Components Palette. But the most important operations you can do with the toolbar are adding, removing, or replacing the icons using the Configure command of the toolbar local menu (simply press the right mouse button over it). This operation invokes the toolbar Editor (see above), one of the Delphi tools with the best user interface, at least in my opinion.

To add an icon to the toolbar, you simply need to find it under the proper category (corresponding to a pull-down menu), and drag it to the bar. In the same way, you can drag an icon away from the toolbar or simply move it to another location. During these operations, you can easily leave some space between groups of icons, to make them easier to remember and select.

The Local Menus

Although Delphi has a good number of menu items, not all of the commands are available though the pull-down menus. At times, you need to use local menus for specific window areas. To activate a local menu, right-click over a window, or press Alt+F10. Even if you have other alternatives, using a local menu is usually faster because you don't need to move the mouse up to the menu bar and select two levels of menus. It's also often easier, since all the local menu commands are related to the current window. Almost every window in Delphi (with the exclusion of dialog boxes) has its own local menu with related commands. I really suggest you get used to right-clicking on windows, because this is not only important in Delphi, but also has become a standard for most applications in Windows. Get used to it, and add local menu to the applications you build with Delphi, too.

Working with the Form Designer

Designing forms is the core of visual development in the Delphi environment. Every component you place on a form and every property you set is stored in a file describing the form (a DFM file) and has some effect on the source code associated with the form (the PAS file).

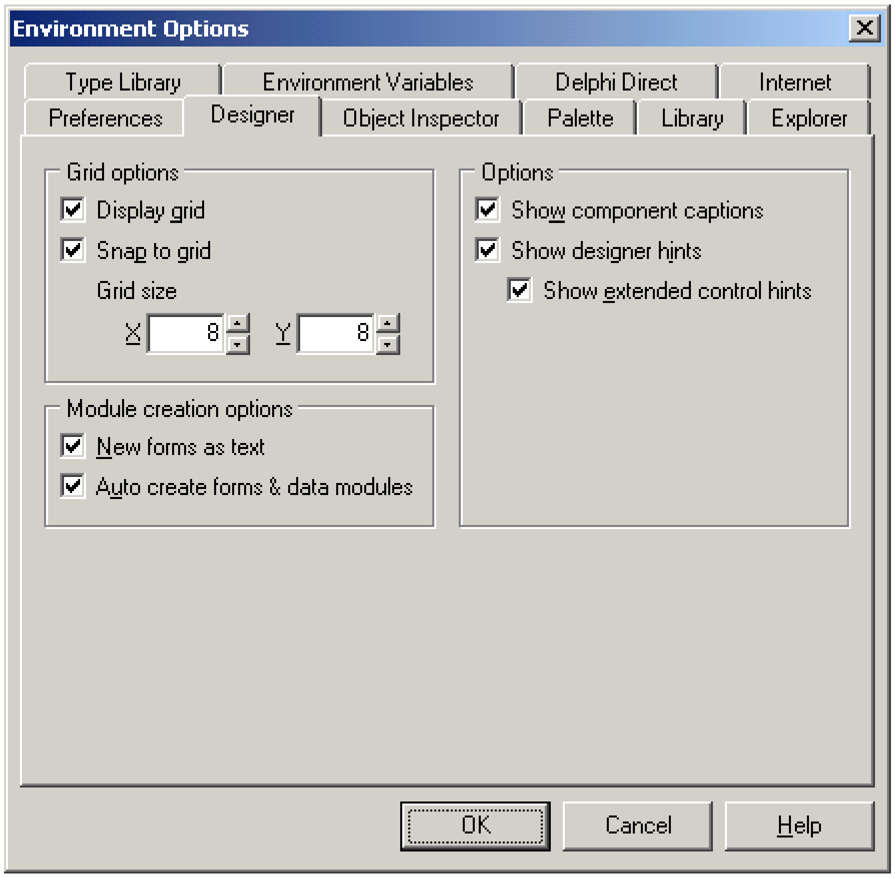
When you start a new, blank project, Delphi creates an empty form, and you can start working with it. You can also start with an existing form (using the various templates available), or add new forms to a project. A project (an application) can have any number of forms. Every time you work with a form at design-time, you are actually using Delphi's Form Designer. When you are working with a form, you can operate on its properties, on the properties of one of its components, or on those of several components at a time. To select the form or a component, you can simply click on it or use the Object Selector (the combo box in the Object Inspector), where you can always see the name and type of the selected item. You can select more than one component by Shift-clicking on the components, or by dragging a selection rectangle around the components on the form.

Even when a component covers the whole surface of the form, you can still select the form with the mouse. Just press and hold Shift while you click on the selected component. This will deselect the component and select the form by default. Using the keyboard, you can press Esc to select the parent of the current component.

While you are working on a form, the local menu has a number of useful features (some of which are also available in the Edit menu). You can use the Bring to Front and Send to Back commands to change the relative position of components of the same kind (you can never bring a graphical component in front of a component based on a window). In an inherited form, you can use the command Revert to Inherited to restore the properties of the selected component to the values of the parent form.

When you have selected more than one component, you can align or size them. Most of the options in the Alignment dialog box are also available in the Alignment palette (accessible through the View | Alignment Palette menu command). You can also open the Tab Order and Creation Order dialog boxes to set the tab order of the visual controls and the creation order of the non-visual controls. You can use the Add to Repository command to add the form you are working on to a list of forms available for use in other projects. Finally, you can use the View as Text command to close the form and open its textual description in the editor. A corresponding command in the editor local menu (View as Form) will reverse the situation.

Along with specific local menu commands, you can set some form options by using the Tools | Options command and choosing the Preferences page (up to Delphi 5) or the Designer page (from Delphi 6). This latter page is shown here:



Grid options
- [✓] Display grid
- [✓] Snap to grid

Grid size X 8 Y 8

Module creation options
- [✓] New forms as text
- [✓] Auto create forms & data modules

Options
- [✓] Show component captions
- [✓] Show designer hints
  - [✓] Show extended control hints

The options related to forms refer to grid activation and size. The grid makes it easier to place components exactly where you want them on the form by "snapping" them to fixed positions and sizes. Without a grid, it is difficult to align two components manually (using the mouse).

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. (The Snap to Grid option works only for mouse operations.) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. If you press Shift+Ctrl+an arrow key, instead, the component will be moved only at grid intervals.

Along with the commands described so far, a form's local menu offers other commands when particular components are selected. In some cases, these menu commands correspond to component properties; others contain particularly useful commands. Table 2.1 lists the commands added to the local menu of a form when some of the components are selected (the TeeChart, Quick Report and Decision Cube components add too many commands to list here). Notice that in some cases these actions are also the default action of the component, the one automatically activated when you double-click on it in the Form Designer.

Table 2.1: Local menu commands added when specific components are selected: [*** not updated for recent versions of Delphi]

Menu Command	**Components**
Menu Designer	MainMenu, PopupMenu

Query Builder	Query (if the Visual Query Builder is available)
Fields Editor	Table, Query, StoredProc, ClientDataSet
Explore	Table, Query, StoredProc, Database
Define Parameters	Query, StoredProc
Database Editor	Database
Assign Local Data	ClientDataSet
UpdateSQL Editor	UpdateSQL
Execute	BatchMove
Columns Editor	DBGrid
Edit Report	Report
ImageList Editor	ImageList
New Page	Page Control
Next/Previous Page	Page Control, NoteBook, TabbedNotebbok
Next/Previous Frame	Animate
Insert Object	OleContainer
New Button	ToolBar
New Separator	
Properties, About	All the ActiveX controls
Action Editor	WebDispatcher
ResponseEditor	QueryTableProducer, DataSetTableProducer

The Component Palette

When you want to add a new component to the form you are working on, you can click on a component in one of the pages of the Component palette, and then click on the form to place the new component. On the form, you can press the left mouse button and drag the mouse to set the position and size of the component at once, or just click to let Delphi use a default size.

Each page of the palette has a number of components; each component has an icon and a name, which appears as a “fly-by” hint (just move the mouse on the icon and wait for a second). The hints show the official names of components, which I’ll use in this book. They are drawn from the names of the classes defining the component, without the initial *T* (for example, if the class is TButton, the name is *Button*).

> If you need to place a number of components of the same kind into a form, shift-click on that component in the palette. Then, every time you click on the form, Delphi adds a new component of that kind. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component palette.

If you are temporarily using a mouse-less computer, you can add a component by using the View | Components List command. Select a component in the resulting list or type its name in the edit window, and then click on the Add to Form button.

Of course, you can completely rearrange the components in the various pages of the palette, adding new elements or just moving them from page to page: select Tools | Options and move to the Palette page. In this page of the dialog box, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page. It’s not a good idea to move components on the palette too often. If you do, you’ll probably waste time trying to locate them afterward.

When you have too many pages in the Component palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: rename the pages with shorter names, so that all the pages will fit on the screen. Obvious...once you've thought about it.

The Object Inspector

When you are designing a form, you use the Object Inspector to set values of component or form properties. Its window lists the properties (or events) of the selected element and their values in two resizable columns. An Object Selector at the top of the Object Inspector indicates the current component and its data type; and you can use it to change the current selection. The Object Inspector doesn't list all of the properties of a component. It includes only the properties that can be set at design-time. As mentioned in Chapter 1, other properties are accessible only at run-time. To know about all the different properties of a component, refer to the Help files.

The right column of the Object Inspector allows only the editing appropriate for the data type of the property. Depending on the property, you will be able to insert a string or a number, choose from a list of options (indicated by an arrow), or invoke a specific editor (indicated by an ellipsis button). When a property allows only two values, such as True and False, you can toggle the value by double-clicking on it. If there are many values available, a double-click will select the next one in the list. If you double-click a number of times, all the values of the list will appear, but it is easier to select a multiple-choice value using the small combo box. For some properties, such as Color, you can enter a value, select an element from the list, or invoke a specific editor! Other properties, such as Font, can be customized either by expanding their sub-properties (indicated by a plus or minus sign next to the name) or by invoking an editor. In other cases, such as with string lists, the special editors are the only way to change a property.

The sub-property mechanism is available with sets and with classes. When you expand sub-properties, each of them has its own behavior in the Object Inspector, again depending on its data type.

You will use the Object Inspector often. It should always be visible when you are editing a form, but it can also be useful to look at the names of components and properties while you are writing code. For this reason, the Object Inspector's local menu has a Stay on Top command, which keeps the Object Inspector window in front of the Form Designer and the editor.

In Delphi 5 and 6 the features of the Object Inspector have been largely extended, with grouping and hiding or little-used properties, interface references, showing of read-only properties, and many other features too complex to discuss here.

The Alignment Palette

The last tool related to form design is the Alignment palette. You can open this palette with the View menu's Alignment Palette command. As an alternative, you can choose the components you want to align, and then issue the Align command from the local menu of the form.

The Alignment palette features a number of commands to position the various controls, center them, space them equally, and so on. To see the effect of each button, simply move the mouse over the window and look at the fly-by hints. When I'm designing complex forms, I position the Alignment palette on the far side of the screen and make sure it always stays in sight by using the Stay on Top command of its local menu.

Writing Code in the Editor

Once you have designed a form in Delphi, you usually need to write some code to respond to some of its events, as we did in Chapter 1. Every time you work on an event, Delphi opens the editor with the source file related to the form. You can easily jump back and forth between the Form Designer and the source code editor by clicking the Toggle Form Unit button on the toolbar, by clicking on the corresponding window, or by pressing the F12 function key.

The Delphi editor allows you to work on several source code files at once, using a "notebook with tabs" metaphor. Each page of the notebook corresponds to a different file. You can work on units related to forms, independent units of Pascal code, and project files; open the form description files in textual format; and even work on plain text files. You can jump from a page of the editor to the next by pressing the Ctrl+Tab keys (or Shift+Ctrl+Tab to move in the opposite direction).

When you work with the editor, you should probably expand its window so that you can see as many full lines of code as possible. A good approach is to size the editor so that it and the Object Inspector are the only windows that appear on the screen when you are writing code. By having the Object Inspector visible you can immediately see the names of the design-time properties of the components.

There are a number of environment options that affect the editor, mostly located in the Editor Options, Editor Display, and Editor Colors pages of the Environment Options dialog box. In the Preferences page, you can set the editor's Autosave feature. Saving the source code files each time you run the program can save the day when your program happens to crash the whole system (something not so rare as you might think). The other three pages of editor options can be used to set the default editor settings like keystroke mappings, syntax highlighting features, and font. Most of these options are fairly simple to understand.

The local menu of the edit window has some commands for debugging and others related to the editor itself, such as those to close the current page, open the file or unit under the cursor, view or hide the message pane below the window, and invoke the editor options discussed before.

Using Editor Bookmarks

The Delphi editor also lets you set line bookmarks. When you are on a line of the editor, you can press Ctrl+Shift plus a number key from 0 to 9 to set a new bookmark, which then appears in the small gutter margin of the editor. Then you can use the Ctrl key plus the number key to jump back at that line of the editor. Pressing the Ctrl+Shift+*number* toggles the status of the bookmark, so you can use this combination again to remove it.

Bookmarks are quite useful when you have a long file and you are editing multiple methods at the same time, or to jump from the class definition to the definition of a method of the class.

Bookmarks have limitation that make them hardly usable. If you set again a given bookmark, the editor moves it. This might seem reasonable, but is actually a problem: If you create a new bookmark and happen to use the number of an existing one, by error, the older bookmark will be removed. Another odd behavior is that you can add multiple bookmarks on the same line, but you'll only see the glyph of one of them. The real problem, however, is that bookmarks are not saved along with the file, nor restored when you reopen it. So they can be used only for a single editing session.

Code Insight

Delphi editor has several features collectively known as Code Insight. The basic idea of this technology is to make it easier for both newcomers and experienced programmers to write code. There are four capabilities that Borland calls Code Insight:

- The Code Completion Wizard 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. It also allows you to look for a proper value in an assignment statement.
- The Code Templates Wizard allows you to insert one of the predefined code templates, such as a complex statement with an inner begin-end block. You can also easily define new templates.
- The Code Parameter Wizard displays, in a hint or ToolTip window, the data type of a function's or method's parameters while you are typing them.
- The ToolTip Expression Evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression under the mouse cursor.

I'll cover the ToolTip Expression Evaluation later in this chapter, while introducing debugging features; in the next three sections I'll give you some more details on the other three Code Insight capabilities. You can enable and disable (or configure) each of these wizards in the Code Insight page of the Environment Options dialog box.

Code Completion

There are two ways to activate this Wizard. You can simply type the name of an object, such as Button1, then add the dot, and wait:

```
Button1.
```

Delphi will display a list of valid properties and methods you can apply to the object. The time you have to wait before the list is displayed depends on the Code Completion Delay option, which you can configure in the Code Insight page.

As an alternative you can type the initial portion of the property or method name, as in Button1.Ca, and then press Ctrl+SpaceBar to get the list immediately, but this time, the Wizard will try to guess which property or method you were looking for by looking at the characters you typed. You can also use this key combination in an assignment statement. If you type:

```
x :=
```

and then press Ctrl+SpaceBar, Delphi will show you a list of possible objects, variables, or constants you can use at this point in the program (that is, in the current scope).

Notice that Delphi determines the elements to show in this list dynamically, by constantly parsing the code you write in the background. So if you add a new variable to a unit, it will show up in the list.

Code Templates

Unlike the Code Completion Wizard, the Code Templates Wizard must be activated manually. You can do this by typing Ctrl+J to show a list of all of the templates.

More often, you'll first type a keyword, such as if or array, and then press Ctrl+J, to activate only the templates starting with those letters. For some keywords Borland has defined multiple templates, all starting with the keyword name (such as *ifA* and *ifB*). So if you press the keyword and then Ctrl+J, you'll get all the templates related to the keyword.

You can also use Code Templates Wizard simply to give a name to a common expression. For example, if you use the MessageDlg function often, you might want to enter a new Code Template called *mess*, type a description, and add then the following text:

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

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

As this example demonstrates, Code Templates have no direct correspondence to language keywords, but are a more general mechanism. Code Templates are saved in the DELPHI32.DCI file, so it should be possible to copy this file to make your templates available on different machines. There seems to be no easy way to merge two Code Templates files, and there are no third-party tools yet to add more templates to a machine. Those enhancements will need to wait until Borland documents the internal structure of the .DCI file.

Code Parameter

The third Code Insight technology I'll discuss here is Code Parameters, the one I was really hoping for and probably like best. Previously when I had to call an unfamiliar function I used to type the name, and then press F1 to jump to the Help system and see its parameters. Now I can simply type the function name, type the open (left) parenthesis, and the parameter names and types appear immediately on a fly-by hint window.

Notice in this figure that the first parameter appear in boldface type. After you type the first parameter and a comma, the second parameter will be set in bold, the same with the third, and so on. This is very useful for functions with many parameters, like some functions of the Window API. Try typing *CreateWindow(* and you'll understand what I mean.

Again, the Code Parameters Wizard works by parsing your code in the background. So if you write the following procedure at the beginning of the implementation section of a unit:

```
implementation

procedure ShowInt (X: Integer);
begin
  MessageDlg (IntToStr (X),
    mtInformation, [mbOK], 0);
end;
```

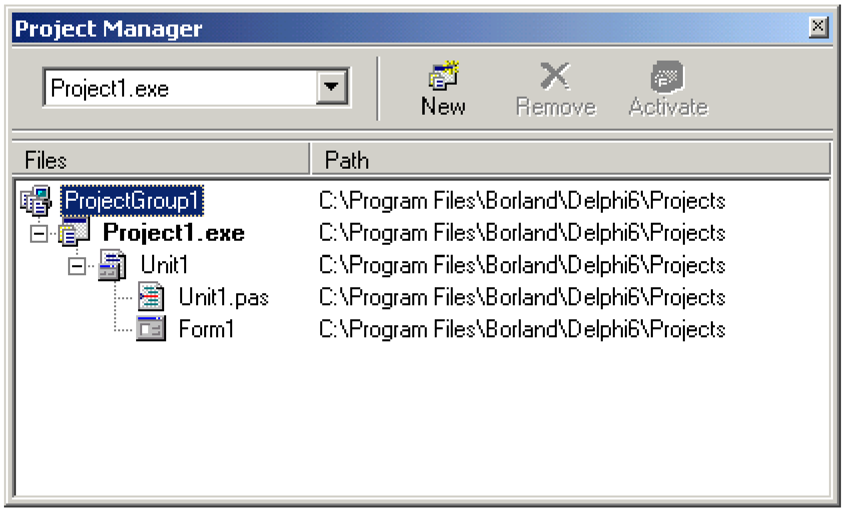
you'll have full information about its parameters when you type *ShowInt(* later.

Managing Projects

In Delphi you also need to know how to manage project files. In Chapter 1, we saw that you can open a project file in the editor and edit the file. However, there are simpler ways to change some of the features of a project. For example, you can use the Project Manager window and Project Options.

The Project Manager

When a project is loaded, you can choose the View | Project Manager command to open a project window. The window lists all of the forms and units that make up the current project. The Project Manager's local menu allows you to perform a number of operations on the project, such as adding new or existing files, removing files, viewing a source code file or a form, and adding the project to the repository. Most of these commands are also available in the toolbar of this window.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi6\Projects
Project1.exe	C:\Program Files\Borland\Delphi6\Projects
Unit1	C:\Program Files\Borland\Delphi6\Projects
Unit1.pas	C:\Program Files\Borland\Delphi6\Projects
Form1	C:\Program Files\Borland\Delphi6\Projects

Setting Project Options

From the Project Manager (or from the Project menu), you can invoke the Project Options dialog. The first page of Project Options, named Forms, lists the forms that should be created automatically at program startup (the default behavior) and the forms that are created manually by the program. You can easily move a form from one list to the other. 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, the other is to set or remove individual options in the source code with the {$X+} or {$X-} commands, where X is the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source code file, or even for just a few lines of code.

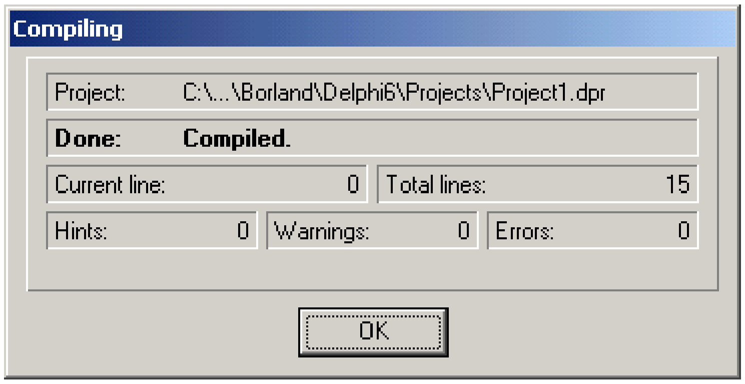
All of the Project Options are saved automatically with the project, but in a separate file with a DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options.

Compiling a Project

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

The project lists the source code files that are part of it, and any related forms. This list is visible both in the project source and in the Project Manager, and is used to compile or rebuild a project. First, each source code file is turned into a Delphi compiled unit, a file with the same name as the Pascal source file and the DCU extension. For example, Unit1.pas is compiled into Unit1.dcu.

When the source code of the project itself is compiled, the compiled units that constitute the project are merged (or linked) into the executable file, together with code from the VCL library. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option. You'll find this option on the Preferences page of the Environment Options dialog box, under the Compiling heading. Although this slows down the compilation a little, the Compile window lets you see which source files are compiled each time (unless your computer is too fast; Delphi might compile several files per second on a fast PC).



Project: C:\...\Borland\Delphi6\Projects\Project1.dpr
Done: Compiled.
Current line: 0 Total lines: 15
Hints: 0 Warnings: 0 Errors: 0

Exploring a Compiled Program

Delphi provides a number of tools you can use to explore a compiled program, including the debugger and the Object Browser.

The Integrated Debugger

Delphi has an integrated debugger with a huge number of features. However, Borland also sells a more powerful stand-alone debugger, called Turbo Debugger. For nearly all of your debugging tasks, the integrated debugger works well enough, particularly if you activate the CPU view window. The stand-alone Turbo Debugger might be useful in a few special cases.

You don't need to do much to use the integrated debugger. In fact, each time you run a program from Delphi, it is executed by default in the debugger. This means that you can set a breakpoint to stop the program when it reaches a specific line of code. For example, open the Hello2 example we created in Chapter 1 and double-click on the button in the form to jump to the related code. Now set a breakpoint by clicking in the editor gutter margin, by choosing the Toggle Breakpoint command of the editor local menu, or by pressing F5.

The editor will highlight the line where you've placed the breakpoint, showing it in a different color. Now you can run the program as usual, but each time you press the button, the debugger will halt the program, showing you the corresponding line of code. You can execute this and the following lines one at a time (that is, step-by-step), look at the code of the functions called by the code, or continue running the program.

When a program is stopped, you can inspect its status in detail. Although there are many ways to inspect a value, the simplest approach is the ToolTip Expressions Evaluation. Simply move the mouse over the name of any variable, and you'll see its current value in a small hint window.

At times the ToolTip Expressions Evaluation seems not to work. This may happen if the optimizing compiler has removed some sections of generated code and placed variables in CPU registers. If you disable the compiler optimizations, you'll get more ToolTips.

The Object Browser

Once you have compiled a program, you can run the Object Browser (available with the View | Browser menu command) to explore it, even if you are not running or debugging it. This tool allows you to see all of the classes defined by the program (or by the units used directly and indirectly by the program), all the global names and variables, and so on. For every class, the Object Browser shows the list of properties, methods, and variables — both local and inherited, private and public. The information displayed in the Object Browser may not mean much if you're still not familiar with the Object Pascal language used by Delphi.

Additional Delphi Tools

Delphi provides many other programming tools. For example, the Menu Designer is a visual tool used to create the structure of a menu. There are also the various Wizards, used to generate the structure of an application or a new form. Other tools are stand-alone applications related to the development of a Windows application, such as the Image Editor and WinSight, a "spy" program that lets you see the Windows message flow.

There are several external database tools, such as the Database Desktop and the Database Explorer (some of which are not available in the lower-level editions of Delphi). A programmer can use other third-party tools to cover weak areas of Delphi. For example, you can use a full-blown resource editor (such as Borland's Resource Workshop), or a tool to generate Help files more easily.

You can also install and use many additional Delphi add-on tools from third-party vendors. You'll find demo versions of some of these tools on the companion CD, but there are many others available. Some of the add-on tools really complement Delphi nicely, and make you more productive.

The Files Produced by the System

As you have seen, Delphi produces a number of files for each project, and you should know what they are and how they are named. There are basically two elements that have an impact on how files are named: the names you give to a project and its forms, and the predefined file extensions used by Delphi for the files you write and those generated by the system.

The great advantage of Delphi over other visual programming environments is that most of the source code files are plain ASCII text files. We explored Pascal source code, project code, and form description files at the end of Chapter 1. Now let's take a minute to look at the structure of options and desktop files. Both types of files use a structure similar to Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of the HELLO.DOF file of the Hello2 example:

```
[Compiler]
A=1
B=0
...
[Linker]
MapFile=0
MinStackSize=16384
MaxStackSize=1048576
...
[Directories]
OutputDir=
SearchPath=
```

In Delphi the option files use the DOF extension, in Kylix the KOF extension (in Delphi 1 they used the OPT extension. These files have different contents and are not fully compatible.

The initial part of this file, which I've omitted, is a long list of compiler options. The same structure is used by the desktop files, which are usually much longer. It is worth looking at what is stored in these files to understand their role. In short, a desktop file (.DSK) lists Delphi windows, indicating their position and status. For example, this is the description of the main window:

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

These are some of the sections related to other windows:

```
[ProjectManager]
[AlignmentPalette]
[PropertyInspector]
[Modules]
[formxxx]
[EditWindowxxx]
[Viewxxx]
```

Besides environment options and window positions, the desktop file contains a number of history lists (lists of files of a certain kind), and an indication of the current breakpoints, watches, active modules, closed modules, and forms.

What's Next

This chapter presented an overview of the Delphi programming environment, including a number of tips and suggestions. Getting used to a new programming environment takes some time, particularly if it is a complex one. I could have devoted this entire book to detailing the Delphi programming environment, but I hope you'll agree that describing how to actually write programs is more useful and interesting.

A good way to learn about the Delphi environment is to use the Help system, where you can look up information about the environment elements, windows, and commands. Spend some time just browsing through the Help files. Of course, the best way to learn how the Delphi environment works is to use it to write programs. That's what Delphi is about. Now we can move on to an important feature of the Delphi environment we have only mentioned: the Object Repository and the Wizards.

Chapter three: The Object Repository and the Delphi Wizards

- Delphi's Object Repository
- Reusing existing applications and forms
- The Database Form Wizard
- Other Delphi Wizards
- Configuring the Object Repository

When you start working on a new application (or simply a new form), you have two choices. You can start from scratch with a blank application or form, or you can choose a predefined model from the Object Repository. If you decide to pick an existing model from the Object Repository, you have even more alternatives. You can make a copy of an existing item (called a template in Delphi 1), you can inherit from an existing item, or you can use one of the available Wizards.

A *Wizard* is a code generator. (Borland used to call them *Experts*, but in Delphi 3 the official name has been changed to Wizards, following Microsoft tradition.) Wizards ask you a number of questions, and use your answers to create some basic code, following predefined rules. You then start working on a project or a form that already has some code and components. Usually, the code generated by these tools can be compiled immediately, and it makes up the basic structure on which you build your program or form.

The purpose of this short chapter is simply to introduce you to the Object Repository and the Delphi Wizards, and to show you how easy they are to use. We won't study the code they generate, since that will be the topic of many examples in the book. From a programming standpoint, the Wizards are really useful. The pitfall is that you might be tempted to use them without trying to understand what they do. For this reason, in some examples I'll build the code manually instead of using the corresponding Wizard.

The Object Repository

Delphi has several 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 menu, and also in other pull-down menus. What happens if you simply select File | New | Other (from Delphi 6, or File | New in earlier versions)? Delphi opens the Object Repository (also called the New dialog box). The Object Repository is used to create new elements of any kind: forms, applications, data modules, libraries, thread objects, components, automation objects, and more. The Object Repository dialog box has a number of pages:

- The *New* and *ActiveX* pages allow you to create many different types of new items. At times when you create a new item, Delphi asks you the name of a new class and few other things, in a sort of *mini-Wizard*.
- The "current project" page (actually you'll see the name of the project) allows you to inherit from a form or data module included in your current project. In this page you can create new

forms or data modules that inherit from those of the current project. The content of this page depends exclusively on the units included in the current project. Simply create a couple of forms and then return to this page, and you'll see that its contents have already changed.

- The *Forms*, *Dialogs*, and *Data Modules* pages allow you to create a new element of these kinds starting from an existing one or using a Wizard.
- The *Projects* page allows you to copy the files from an existing project stored in the Repository, or use the Application Wizard.
- The *Multitier, Business, WebSnap,* and *WebServices* pages provide a starting point for using some of the advanced features found only in the Enterprise edition (and mostly only starting with Delphi 6)

Use the radio buttons at the bottom of the Object Repository dialog box to indicate that you want to copy an existing item, inherit from it, or use it directly without making a copy.

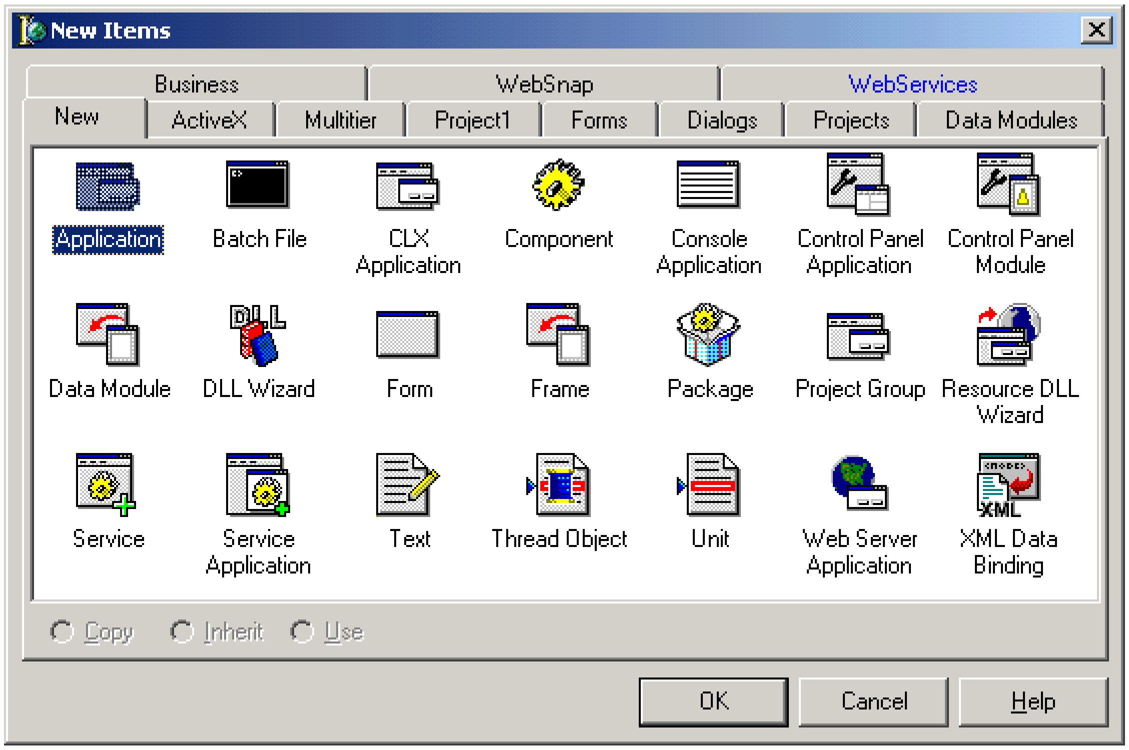
The concept of inheritance in object-oriented programming is discussed in Mastering Delphi. In short, it is a way to add new capabilities to an existing form (or class in general) without making a full copy. This way, if you make a change in the original form (or class), the inherited form (or class) will be affected, too.

When you select a Wizard instead of a template, the only available radio button is *Copy*, meaning you'll end up with a new copy, the generated code. Keep in mind that I am discussing the pages of the Object Repository as they appear in Delphi 6 Enterprise, but different editions and versions of Delphi have fewer or different pages and items; and you can further customize the Repository, as we will see later on in this chapter.

The Object Repository has a local menu that allows you to sort items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, details). This last view is the only one that gives you the description, the author, and the date of the tool. This information is particularly important when looking at Wizards, projects, or forms you've added to the Repository.

The New Page

The New page of the Object Repository (shown below) allows you to create several new items of the more commonly used kinds and is often an alternative to a direct menu command. Here is a list of the elements you can create from this page:



- *Application* creates a new blank project (the same as the command File | New | Application).
- *Batch File* opens a batch file you can include in a project group for automatic processing (to fully automate a complex build and deploy process).
- *CLX Application* creates a new blank project based on the CLX library (and on Qt), so that it will be fully portable to Kylix on the Linux platform (the same as the command File | New | CLX Application).
- *Component* creates a new Delphi component after you've completed the information requested by the simple Component Wizard. The same wizard can be activated with the Component | New Component menu command.
- *Console Application*
- *Control Panel Application*
- *Control Panel Module*
- *Data Module* creates a new blank data module (the same as the command File | New Data Module).
- *DLL Wizard* creates a simple DLL skeleton.
- *Form* creates a new blank form (the same as the command File | New Form).
- *Frame*
- *Package* creates a new Delphi component package. (You can also create a new package when creating a component.)

- *Project Group*
- *Resource DLL Wizard*
- *Service*
- *Service Application*
- *Text* opens a new ASCII text file in the Delphi editor.
- *Thread Object* creates a new thread object after asking you to fill in the New Thread Object dialog box. Multithreading in Windows is introduced in Chapter 25.
- *Unit* creates a new blank unit, a Pascal source file not connected with a form.
- *Web Server Application* allows you to create an ISAPI/NSAPI add-in DLL, a CGI stand-alone executable, a Win-CGI stand-alone executable, an Apache module, or a server application based on Delphi's Web Debugger. In each case Delphi creates a simple project based on a Web module (a special type of data module) instead of a form.
- *XML Data Binding*

*[***Many of the descriptions for newer entries are still missing]*

As some of the features made available by the Object Repository are quite advanced, it doesn't make

The Forms Page

This page lists predefined forms. Here is a short list of the predefined forms available in Delphi:

- *About box* is a simple About box.
- *Dual list box* is a form with two different list boxes, allowing a user to select a number of elements from one list and move them to the other list by pressing a button. Along with the components, this form contains a good amount of nontrivial Pascal code.
- *QuickReport Labels* creates a report form based on the QuickReport component.
- *QuickReport List* creates another form based on QuickReport, with a different layout.
- *QuickReport Master/Detail* is a third predefined report form with a more complex structure.
- *Tabbed pages* is a form based on the Windows PageControl.

Wizards can only be executed. The other forms, by contrast, can be used in multiple ways. You can add them into a project, inherit from them (in this case the original form will be automatically added to your project), or use them directly. When you use a form or inherit from one, take care not to make any change on the original forms in the Repository.

The Dialogs Page

This page is similar to the previous one, but includes forms that are typically used as dialog boxes. Here is the list of its items:

- *Dialog with help* is available in two versions. One has the buttons on the right side of the form (*Vertical*), and the other has them in the lower portion (*Horizontal*), as you can see from the corresponding icons.
- *Dialog Wizard* is a simple Wizard capable of generating different kinds of dialog boxes with one or more pages, as we will see later in this chapter.
- *Password dialog* is a dialog box with a simple edit box for entering a password.
- *Reconcile Error Dialog* is used in MIDAS or DataSnap applications.
- *QuickReport Wizard* creates a Print Options dialog box for a report.
- *Standard dialog* is also available in two versions, again *Horizontal* and *Vertical*, with buttons in different positions.

The Data Modules Page

You already know what a project and a form are, but what is a data module? It is a sort of form that never appears on screen at run-time and can be used to hold nonvisual components. It is mostly used to implement code related to database access. This page has only a data module at start-up, *Customer Data*. If you have several forms or applications accessing the same data tables and database queries, you can easily define new data modules and add them to the repository.

The Projects Page

The last page we will look at in this introduction contains project schemes you can use as the starting point for your own application. These projects often include one or more forms. Here is the list of them:

- *Application Wizard* is another simple Wizard that allows you some limited choices for the file structure and other elements of an application.
- *MDI Application* defines the main elements of a Multiple Document Interface (MDI) program. It defines a main form for the MDI frame window, with a menu, a status bar, and a toolbar. It also defines a second form that can be used at run-time to create a number of child windows.
- *SDI Application* defines a main form with the standard attributes of a modern user interface, including a toolbar and a status bar, and also a typical About box.
- *Win2000 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 2000 compatibility logo. This command basically creates an SDI application with a RichEdit component in it, and adds the code needed to make the application mail-enabled.

- *Win95 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 95/98 compatibility logo. The application is similar to the one above.

When you select one of these projects, Delphi asks you to enter the name of an existing or a new directory. If you indicate a new directory, Delphi will automatically create it.

For example, we can create an SDI project based on the corresponding template. Then we can customize it, giving it a proper title, removing or adding new components, and writing some code. Some interesting components are already there, however, and there is even some ready-to-use code to make those components work properly. The menu and toolbar of the application, can be used to open some dialog boxes. File Open and File Save dialog boxes are wrapped up in components that are added to the form by the template; the About box is defined by the template as a second form.

In the simple SdiTemp example, I've decided to make just a few limited changes: I've entered a new title for the main form and some text for the labels for the About box (the property to use is Caption in both cases). The result is the application shown in *[***missing figure]* and available in the source code.

Delphi Wizards

Besides copying or using existing code, Delphi allows you to create a new form, application, or other code files, using a Wizard. Wizards (or Experts) allow you to enter a number of options and produce the code corresponding to your choices.

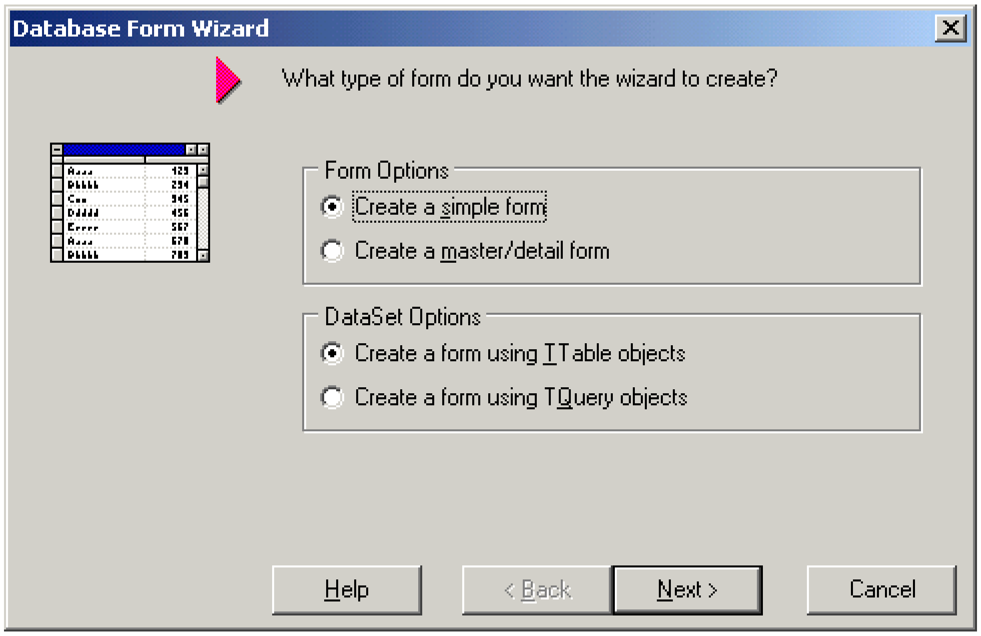
One of the most important predefined wizards is the Database Form Wizard, which you can activate using the Database | Form Wizard menu item or the icon in the Forms page of the Object Repository. There are also some other simple Wizards in Delphi, such as the Application Wizard and the Dialog Wizard. I've listed various other Wizards in the description of the pages of the Object Repository, in the last section.

You can also buy add-on Wizards from third-party tool providers, download one from my web site, or even write your own Wizard. Add-on Wizards often show up in the Help menu, but it is possible to add new menu items in other Delphi pull-down menus or install Wizards in various pages of the Object Repository.

The Database Form Wizard

In this section, I'll show you a quick example of the use of the Database Form Wizard, but I won't describe the application we build in detail. In this example, we'll build a database program using some of the data already available in Delphi. Note that you have to create a project first, and then start the Database Form Wizard. So you usually end up with two forms, unless you remove the original form from the project. Fortunately, one of the Wizard's options, displayed at the end, lets you select the new form generated by the Wizard as the main form.

1. As soon as you start the Database Form Wizard, you will be presented with a number of choices, which depend on the options you choose at each step. The first page lets you choose between a simple or a master detail form, and between the use of tables or queries. Leave the selections as they are by default, and move on by clicking on the Next button.

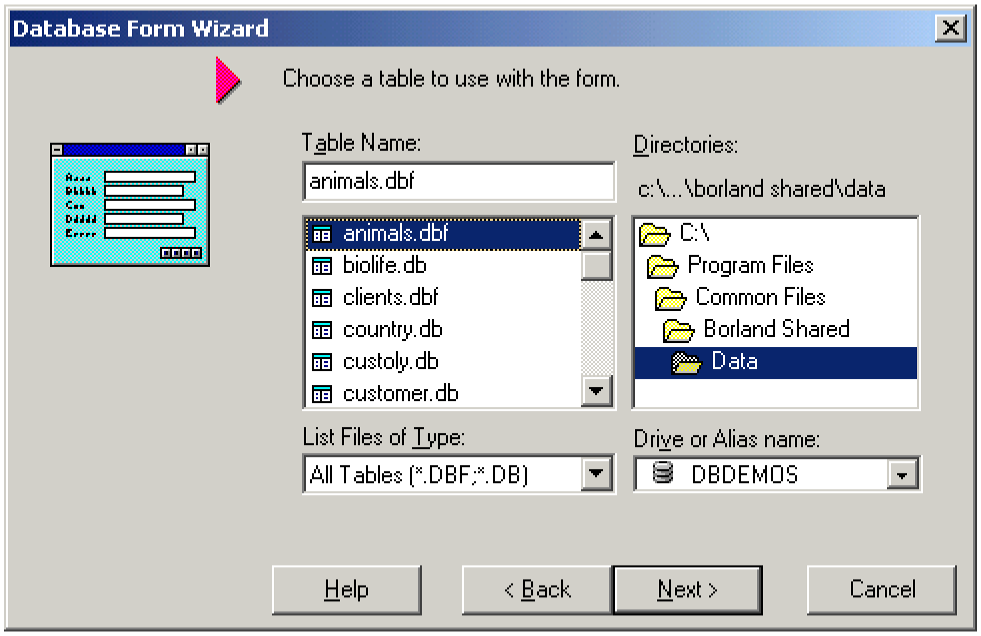

Database Form Wizard

What type of form do you want the wizard to create?

Form Options
[✓ Create a simple form]
Create a master/detail form

DataSet Options
[✓ Create a form using TTable objects]
Create a form using TQuery objects

- **2.** In the next page you can choose an existing database table to work on. In the Drive or Alias Name combo box, there should be a DBDEMOS alias. After you select this option, a number of Delphi demo database tables appear in the list. Choose the first, `animals.dbf`.


Database Form Wizard

Choose a table to use with the form.

Table Name: animals.dbf

animals.dbf
biolife.db
clients.dbf
country.db
custoly.db
customer.db

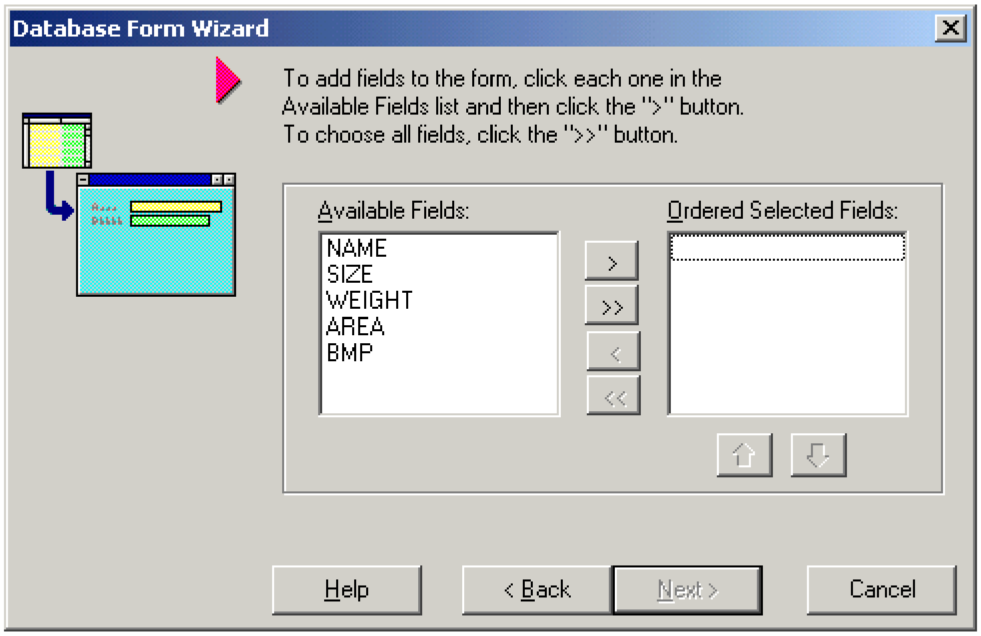
Directories: c:\...\borland shared\data

C:\
Program Files
Common Files
Borland Shared
Data

List Files of Type: All Tables (*.DBF;*.DB)

Drive or Alias name: DBDEMOS

- **3.** In the third page, you can choose the fields of the selected database table that you want to consider. To build this first example, choose all of the fields by clicking on the >> button.

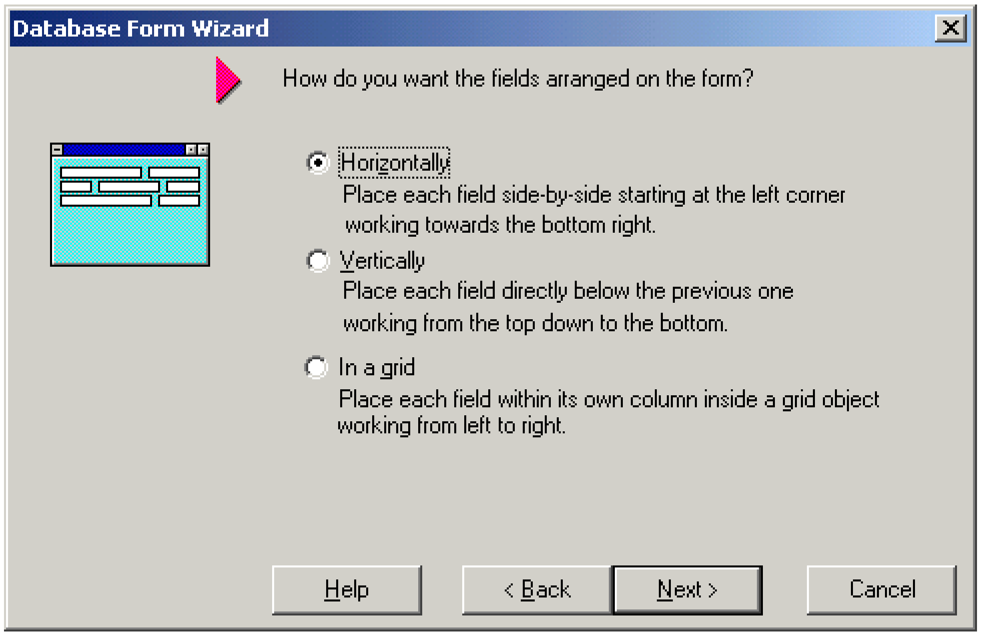


To add fields to the form, click each one in the Available Fields list and then click the ">" button. To choose all fields, click the ">>" button.

NAME
SIZE
WEIGHT
AREA
BMP

•

4. On the next page, you can choose from various layouts. If you choose Vertical, the next page will ask you the position of the labels. The default option, Horizontal, might do.



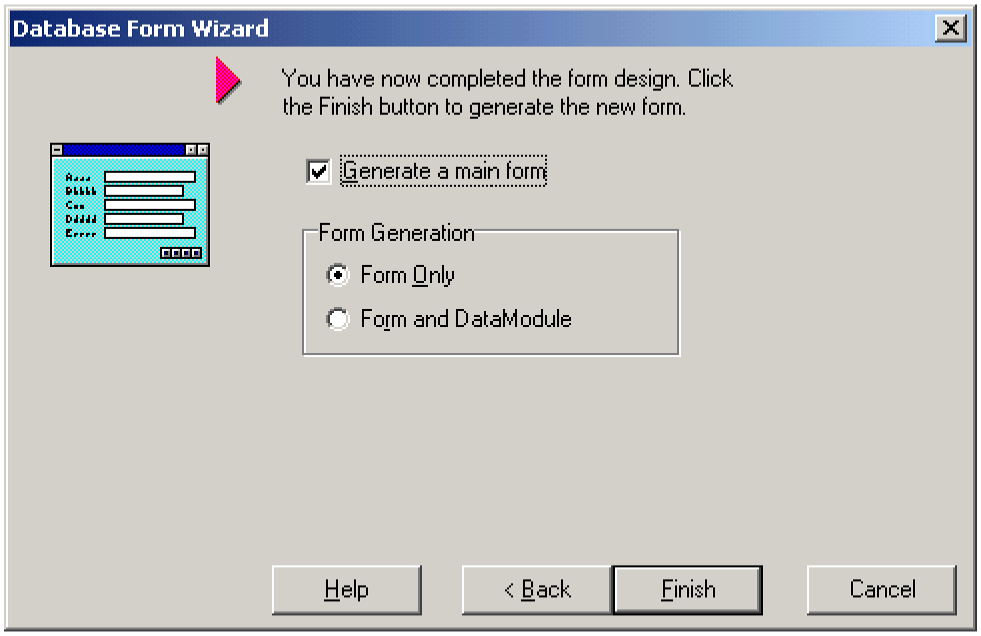
How do you want the fields arranged on the form?

[✓ Horizontally]
Place each field side-by-side starting at the left corner working towards the bottom right.

Vertically
Place each field directly below the previous one working from the top down to the bottom.

In a grid
Place each field within its own column inside a grid object working from left to right.

• **5.** The next page is the last. Leave the Generate a Main Form check box and the Form Only radio button selected, and click on the Finish button.


Database Form Wizard

You have now completed the form design. Click the Finish button to generate the new form.

[✓] Generate a main form

Form Generation

(●) Form Only

( ) Form and DataModule

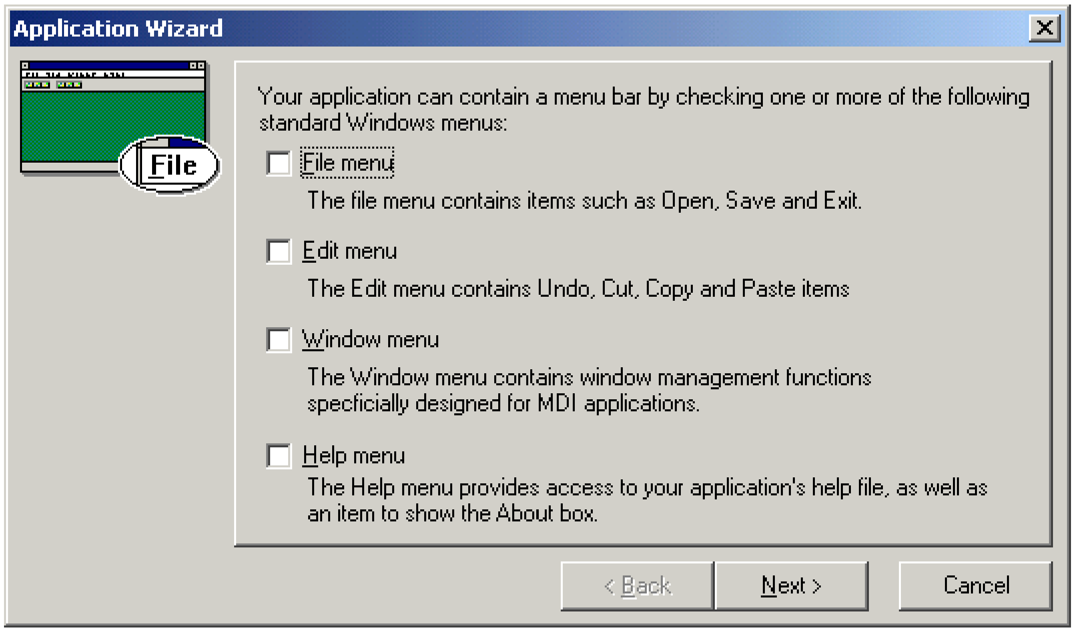
You can immediately compile and run the application. The result is a working database application, which allows you to navigate among many records using the buttons. This specific application (the DataExp example) even has a graphical field, displaying a bitmap with the current animal.

The output of the generated form is usually far from adequate. In this case, the image area is too small; at other times the positioning of the controls may not be satisfactory. Of course, you can easily move and resize the various components placed on the form at design-time.

To make this task easier, you can select the Table component (the one in the upper-left corner of the form) and toggle its Active property to True. Then the data of the table's first record will be displayed at design-time. This is helpful because it allows you to see an example of the length of the field's text and the size of the image. Note that the Database Form Wizard generates almost no Pascal code, besides a line used to open the table when the program starts. The capabilities of the resulting programs stem from the power of the database-related components available in Delphi.

The Application Wizard

Another interesting (although less powerful) tool is the Application Wizard. You can activate it from the Projects page of the Object Repository. The Application Wizard allows you to create the skeleton of a number of different kinds of applications, depending on the options you select.



Your application can contain a menu bar by checking one or more of the following standard Windows menus:

☐ File menu
The file menu contains items such as Open, Save and Exit.

☐ Edit menu
The Edit menu contains Undo, Cut, Copy and Paste items

☐ Window menu
The Window menu contains window management functions specficially designed for MDI applications.

☐ Help menu
The Help menu provides access to your application's help file, as well as an item to show the About box.

The first page of this Wizard allows you to add some standard pull-down menus to the program: File, Edit, Window, and Help. If you select the File menu, the second page will ask you to enter the file extensions the program should consider. You should enter both a description of the file, such as *Text file (*.txt)*, and the extension, *txt*. (You can input several extensions, each with its own description.) These values will be used by the default File Open and File Save dialog boxes that the Application Wizard will add to the program if you select the file support option.

Then, if you have selected any of the pull-down menus, the Application Wizard displays a nice visual tool you can use to build a toolbar. Unfortunately, this tool is not available as a separate editor inside Delphi. You simply select one of the pull-down menus, and a number of standard buttons corresponding to the typical menu items of this pull-down menu appear (but only if the menu has been selected on the first page of the Application Wizard).

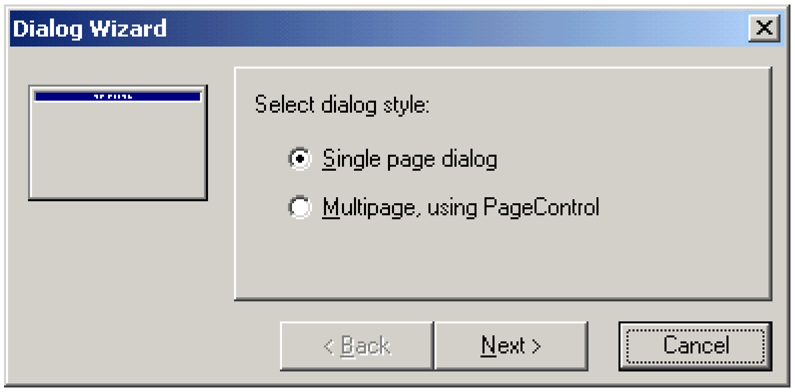
To add a new button, select one of them in the graphical list box on the right and press the Insert button. The new toolbar button will be added at the position of the small triangular cursor. You can move this cursor by clicking on one of the elements already added to the toolbar. This cursor is also used to indicate a button you want to remove from the toolbar.

When the toolbar is finished, you can move to the last page. Here you can set several other options, such as choosing MDI support, adding a status bar, and enabling hints. You can also give a name to the new application and specify a directory for the source files. The name of the application can be long, but it cannot contain white spaces (it should be a valid Pascal identifier), and the directory for the application should be an existing directory. To place the project files in a new directory, choose the Browse button, enter the new path, and the dialog box will prompt you to create the new directory.

Although it is somewhat bare and it has room for improvement, the Delphi Application Wizard is much more useful than the predefined application templates for building the first version of an application. One of its biggest advantages is that you can define your own toolbar. Another advantage is that the Application Wizard generates more code (and more comments) than the corresponding templates do. The disadvantage of this Wizard is that it generates an application with a single form. Its MDI support is limited, because no child form is defined, and the generated application has no About box.

The Dialog Wizard

Delphi's Dialog Wizard is a simple Wizard provided mostly as a demo, with its own source code. From the code of this Wizard, in theory you should be able to learn how to build other Wizards of your own. However, you can still use the Dialog Wizard as a tool to build two kinds of dialog boxes: simple dialog boxes and multiple-page dialog boxes based on the Windows PageControl component.



Select dialog style:

(●) Single page dialog
( ) Multipage, using PageControl

If you choose the simple dialog box, the Wizard will jump to the third page, where you can choose the button layout. If you choose the multiple-page dialog box, an intermediate page will appear to let you input the text of the various tabs. This Wizard is an alternative to the corresponding form templates of the Object Repository. Its advantage is that it allows you to input the names of the PageControl tabs directly.

Customizing the Object Repository

Since writing a new wizard is far from simple, the typical 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 New Application Templates

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 two or more similar programs, you can save the current status to a template, ready to use later on.

Although Borland calls everything you can put in the Object Repository an *object*, from an object-oriented perspective this is far from true. For this reason I call the schemes you can save to disk for later use *templates*. Application templates, in particular, do not relate to objects or classes in any way, but are copied to the directory of your new project. *Object Repository* sounds much better than *Browse Gallery* (the name used in Delphi 1), but besides the capability to activate form inheritance, there is not much object-oriented in this tool.

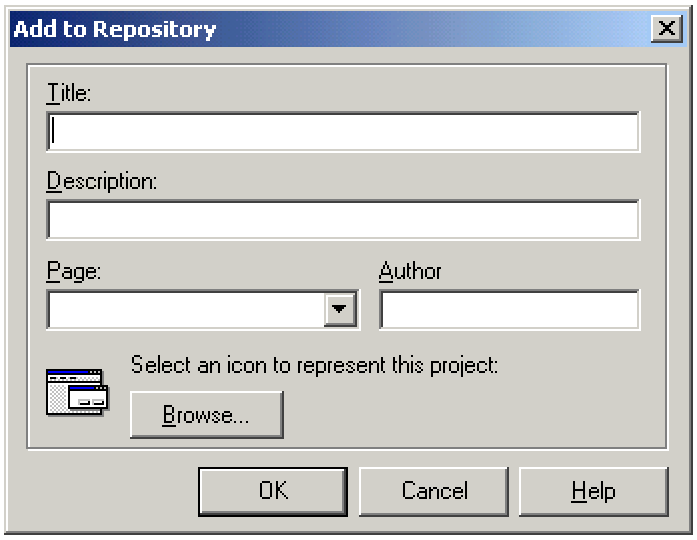
You can add a project to the Repository by using the Project | Add to Repository command, or by using the corresponding item of the local menu of the Project Manager window. As a very simple example, just to demonstrate the process, the following steps describe how you can save the slightly modified version of the default SDI template (shown earlier) as a template:

1. Open the modified SdiTemp example (or any other project you are working on).

2. Select the Project | Add to Repository menu command (or the Add Project to Repository command in the local menu of the Project Manager window).

3. In the Add to Repository dialog box, enter a title, a description for the new project template, and the name of the author. You can also choose an icon to indicate the new template or accept the default image. Finally, choose the page of the Repository where you wish to add the project.

4.: Click on OK, and the new template is added to the Delphi Object Repository.



Now, each time you open the Object Repository, it will include your custom template. If you later discover that the template is not useful any more, you can remove it. You can also use a project template to make a copy of an existing project so that you can continue to work on it after saving the original version.

However, there is a simpler way to accomplish this: copy the source files to a new directory and open the new project. If you do copy the source files, do *not* copy the DSK file, which indicates the position of the Windows on the screen. The DSK file holds a list of files open in the editor, using an absolute path. This means that as soon as you open the new project and start working on it, you may well end up editing the source code files of the original project and compiling the files of the new version (the project manager stores relative paths).

This will certainly surprise you when the changes you make in code or in forms seem to have no effect. Simply deleting the DSK file, or not copying it in the first place, avoids this problem.

The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or wizards, this is not what you want. 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.

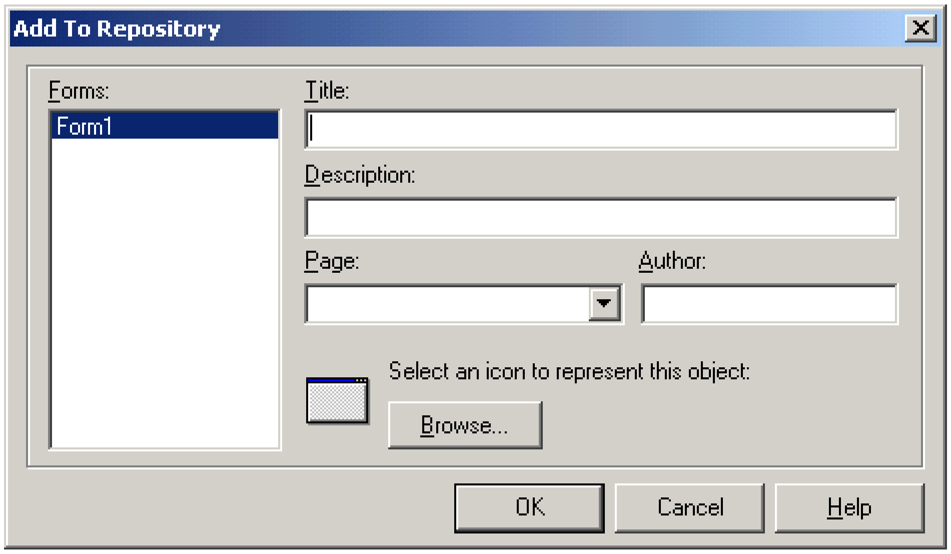
2: Remove its only form from the project.

3: Add this project to the templates, naming it *Empty Project*.

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 need to use the File | Save Project As command to give a new name to the project, since saving the project automatically uses the default name in the template.

Adding New Form Templates to the Object Repository

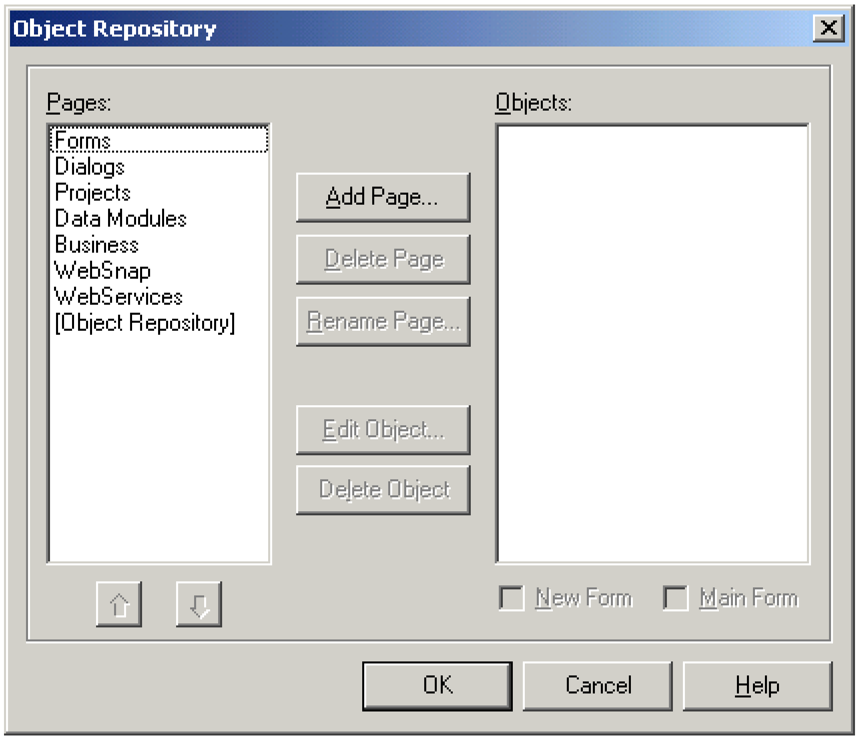
Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form you want to add, right-click on it, and select Add to Repository from the local menu. In the dialog box that appears (see below), you can choose which form of the current project should be added to the Repository, and set the title, description, author, page, and icon, as usual. Once you have set these elements and clicked on OK, the form is added to the proper page of the Object Repository.



This approach is suggested if you have a complex form to make available to other applications and other programmers. They will be able to use your form as is, make a copy of it, and inherit from it. For this reason, adding forms to the Repository is far more flexible than adding projects, which can only be copied as the starting point of a new application.

The Object Repository Options

To further customize the Repository, you can use the Tools | Repository command to open the Object Repository dialog box. This dialog box is quite easy to use; on the left is the list of current Repository pages and on the right the list of items in each page, including both templates and Wizards. You can use this dialog to move repository items to different pages, to add new elements, or to delete existing ones.



You can use the three page-related buttons and the two buttons with arrows below the list of pages to arrange the structure of the Object Repository, adding new pages, renaming or deleting them, and changing their order. All these operations affect some of the tabs of the Object Repository itself (other tabs are fixed).

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 the current form or Wizard as the default, to be used when a new form is created (File | New | Form). Only one object in the Object Repository can be used as a default; it is marked with a special symbol placed over its icon.

- The *Main Form* check box, instead, is used to indicate the form or Wizard used to create the main form of a new application (File | New | Application) when no special New Project is selected (see next bullet). The current Main form is indicated by a second special symbol.
- The New Project check box, available when you select a project object, can be used to mark the default project that Delphi uses when you issue the File | New | Application command. Also, the New Project is indicated by its own special symbol.

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 directly use a form or any other object without copying it, then you end up having some files of your project 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. Wizards may be part of components or packages, and in this case are installed the same way you install a component or a package. Other Wizards are distributed as stand-alone DLLs. In this case you should add the name of the DLL in the Windows Registry under the key:

Software\Borland\Delphi x.0\Experts.

Simply add a new string key under this, choose a name you like (it doesn't really matter) 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.

What's Next

In this short chapter, we have seen how you can start the development of an application by using Delphi templates and Wizards. You can use one of the predefined application templates or form objects, start the Database Form Wizard, or use the other Wizards for a fast start with applications, forms, and other objects.

However, I'll rarely use these templates and Wizards in the rest of the book. With the exception of the Database Form Wizard, these tools let you build only very simple applications, which you can often put together yourself in seconds when you are an experienced Delphi programmer. For beginners, Wizards and templates provide a quick start in code development. But you are not going to *remain* a beginner, are you?

So with the next chapter we'll start focusing on the use of the actual components and controls available in Delphi, to build actual applications, even if simple ones. Differently form other books of mine, which focus more on the foundations, this text proceeds mainly though practical examples.

Notice that the only reason there is no introduction to the use of the language and the core structure of the VCL, is that because you can find these topics covered in Essential Pascal (for the core language) and Mastering Delphi (for the OOP features and the VCL architecture).

Chapter 4: A Tour of the Basic Components

- Clicking a button or another component
- Adding colored text to a form
- Dragging from one component to another
- Accepting input from the user
- Creating a simple editor
- Making a choice with radio buttons and list boxes
- Allowing multiple selections
- Choosing a value in a range

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the Visual Component Library, we are ready to delve into the central part of the book: the use of components. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

The system comes with a number of ready-to-use components. I will not describe every component in detail, examining each of its properties and methods. If you need this information, you can find it easily in the Help system. The aim of Part II of this book is to show you how to use some of the features offered by the Delphi predefined components to build applications. In fact, this chapter and those following will be based heavily on sample programs. These examples tend to be quite simple — although I've tried to make them meaningful — in order to focus on only a couple of features at a time.

I'll start by focusing on a number of basic components, such as buttons, labels, list boxes, edit fields, and other related controls. Some of the components discussed in this chapter are present in the Standard page of the Delphi Components palette; others are in different pages. I'm not going to describe all the components of the Standard page, either. My approach will be to discuss, in each chapter, logically related components, ignoring the order suggested by the pages of the Components palette.

Windows Own Components

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window with a specific behavior, some properties, and some methods (although traditional C language code used to access the predefined components in Windows by sending and receiving messages). These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components.

Actually Microsoft's third step is its ActiveX controls (an extension of the older OCX technology), the natural successor of VBX controls. In Delphi you can use both ActiveX and native components, but if you look at the technology, Delphi components are really ahead of the ActiveX controls. It is enough to say that Delphi components use OOP to its full extent, while ActiveX controls do not fully implement the concept of inheritance. With the recent introduction of the .NET framework, Microsoft has finally followed Borland and Sun (with Java) with the adoption of a technology that treats components as classes and vice verse.

Windows 3 had six kinds of predefined controls, generally used inside dialog boxes. They were buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Windows 95 added a number of new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. These controls were already used by programmers, who had to re-implement them each time. Windows 95 developers could for the first time use the standard common controls provided by the system, and starting with Delphi 3 developers had the advantage of having corresponding easy-to-use components.

The standard system controls are the basic components of each Windows application, regardless of the programming language used to write it, and are very well known by every Windows user. Delphi literally wraps these Windows predefined controls in some of its basic components — including those discussed in this chapter.

Notice that CLX provides components similar to Windows common controls that are portable to Kylix. This is important as Linux, of course, hasn't got Microsoft's common controls!

Clicking a Button

In the first chapter of this book, we built small applications based on a button. Clicking on that button caused a "*Hello*" message to appear on the screen. The only other operation that program performed was moving the button so that it always appeared in the middle of the form. Since then we've seen other examples that used buttons as a way to perform a given action (using their `OnClick` event).

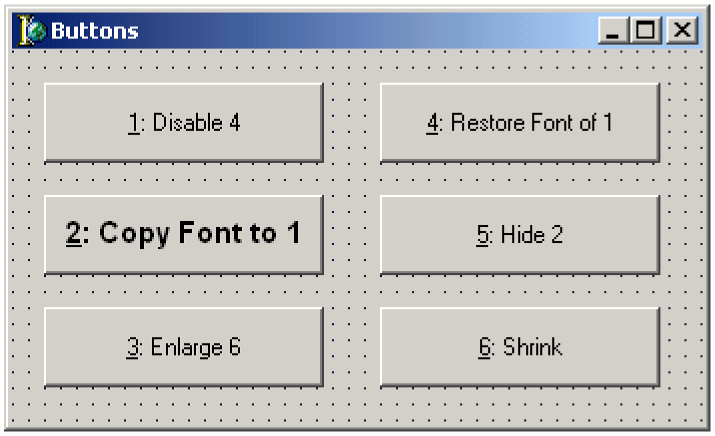
Now we are going to build another form with several buttons and change some of their properties at run-time; in this example, clicking a button will usually change a property of another button. To build the Buttons program, I suggest you follow the instructions closely at first and then make any changes you want. Of course, you can read the description in the book and then work on the source files, if you prefer.

The Buttons Example

First, open a new project and give it a name by saving it to disk. I've given the name `ButtonF.pas` to the unit describing the form and the name `Buttons.dpr` to the project. Now you can create a number of buttons, let's say six.

Instead of selecting the component, dragging it to the form, and then selecting it again to repeat the operation, you can take a shortcut. Simply select the component by clicking on the Components palette while holding down Shift. The component will remain selected, as indicated by a little border around it. Now you can create a number of instances of that component.

Even if you use the grid behind the form, you might need to use the Edit | Align command to arrange the buttons properly. Remember that to select all six buttons at a time, you can either drag a rectangle around them or select them in turn, holding down the Shift key. In this case, it's probably better to select a column of three buttons at a time and arrange them.



Now that we have a form with six buttons, we can start to set their properties. First of all, we can give the form a name (`ButtonsForm`) and a caption (*Buttons*). The next step is to set the text, or `Caption` property, of each button. Usually, a button's `Caption` describes the action performed when a user clicks on it. We want to follow this rule, adding the number of the button at the beginning of each caption. So if the first button disables button number four (which is the one on the same row), we can name it *1: Disable 4*. Following the same rule, we can create captions for the other buttons.

For a summary of the properties of the components of this form, you can refer to its textual description:

```
object ButtonsForm: TButtonsForm
  Caption = 'Buttons'
  object Button1: TButton
    Caption = '&1: Disable 4'
    OnClick = Button1Click
  end
  object Button2: TButton
    Caption = '&2: Copy Font to 1'
    Font.Color = clBlack
    Font.Height = -15
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ParentFont = False
    OnClick = Button2Click
  end
  object Button3: TButton
    Caption = '&3: Enlarge 6'
    OnClick = Button3Click
  end
  object Button4: TButton
    Caption = '&4: Restore Font of 1'
    OnClick = Button4Click
  end
  object Button5: TButton
    Caption = '&5: Hide 2'
    OnClick = Button5Click
  end
  object Button6: TButton
    Caption = '&6: Shrink'
    OnClick = Button6Click
  end
end
```

Notice that every button has an underlined shortcut key, in this case the number of the button. Simply by placing an ampersand (&) character in front of each caption, as in *'&1: Disable 4'*, we can create buttons that can be used with the keyboard. Just press a number below 7, and one of the buttons will be selected, although you won't see it pressed and released.

The final step, of course, is to write the code to provide the desired behavior. We want to handle the OnClick event of each button. The easiest code is that of Button2 and Button4. When you press Button2, the program copies the font of this button (which is different from the standard font of the other buttons) to Button1, and then disables itself:

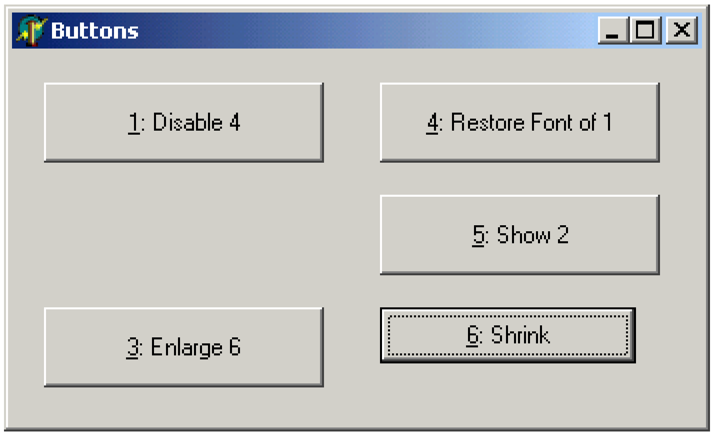
```
procedure TButtonsForm.Button2Click(Sender: TObject);
begin
  Button1.Font := Button2.Font;
  Button2.Enabled := False;
end;
```

Pressing Button4 restores the original font of the button. Instead of copying the font directly, we can restore the font of the form, using the ParentFont property of the button. The event also enables Button2, so that it can be used again to change the font of Button1:

```
procedure TButtonsForm.Button4Click(Sender: TObject);
begin
  Button1.ParentFont := True;
  Button2.Enabled := True;
end;
```

To implement the Disable and Hide operations of Button1 and Button5, we might use a Boolean variable to store the current status. As an alternative, we can decide which operation to perform while checking the current status of the button — the status of the Enabled property. The two methods use two different approaches, as you can see in the following code:

```
procedure TButtonsForm.Button1Click(Sender: TObject);
begin
  if not Button4.Enabled then
  begin
    Button4.Enabled := True;
    Button1.Caption := '&1: Disable 4';
  end
  else
  begin
    Button4.Enabled := False;
    Button1.Caption := '&1: Enable 4';
  end;
end;

procedure TButtonsForm.Button5Click(Sender: TObject);
begin
  Button2.Visible := not Button2.Visible;
  if Button2.Visible then
    Button5.Caption := '&5: Hide 2'
  else
    Button5.Caption := '&5: Show 2';
end;
```



You can see the results of this code in the figure above. The last two buttons have *unconstrained* code. This means that you can shrink `Button6` so much that it will eventually disappear completely:

```
procedure TButtonsForm.Button3Click(Sender: TObject);
begin
  Button6.Height := Button6.Height + 3;
  Button6.Width := Button6.Width + 3;
end;

procedure TButtonsForm.Button6Click(Sender: TObject);
begin
  Button6.Height := Button6.Height - 3;
  Button6.Width := Button6.Width - 3;
end;
```

It would have been quite easy, in any case, to check the current size of the button and prevent its reduction or enlargement by more than a certain value.

Clicking the Mouse Button

Up to now we have based the examples on the `OnClick` event of buttons. Almost every component has a similar click event. But what exactly is a click? And how is it related to other events, such as `OnMouseDown` and `OnMouseUp`?

First, consider the click. At first sight you might think that to generate a click, a user has to press and release the left mouse button on the control. This is certainly true, but the situation is more complex. When the user clicks the left mouse button on a button component, the component is graphically pressed, too. However, if the user moves the cursor (holding down the left mouse button) outside the button surface, this button will be released. If the user now releases the left mouse button outside the button area, no effect — no click — takes place. On the other hand, if the user places the cursor back on the button, it will be *pressed* again, and when the mouse button is released, the click will occur. If this is not clear, experiment with a button; any button in any Windows application will do.

Now to the second question. In Windows, the behavior just described is typical of buttons, although Delphi has extended it to most components, as well as to forms. In any case, the system generates more basic events — one each time a mouse button is pressed, and another each time a button is released. In Delphi, these events are called `OnMouseDown` and `OnMouseUp`. Since the mouse has more than one button, these same events are generated when the user presses any mouse button.

You might want different actions to occur, depending on the mouse button. For this reason, these event handlers include a parameter indicating which button was pressed. These methods also include another parameter, indicating whether some special key (such as Shift or Ctrl) has been pressed and, finally, two more values indicating the *x* and *y* positions where the action took place. This is the method corresponding to this event for a form:

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

Most of the time, we do not need such a detailed view, and handling the mouse-click event is probably more appropriate.

Adding Colored Text to a Form

Now that you have played with buttons for a while, it's time to move to a new component, labels. Labels are just text, or comments, written in a form. Usually, the user doesn't interact with a label at all — or at least not directly. It doesn't make much sense to click on a label (although in Delphi this is technically possible). Keep in mind, however, that not all of the text you see in a form corresponds to a label. A form (and any other component) can simply output text on its surface, for example using the `TextOut` method.

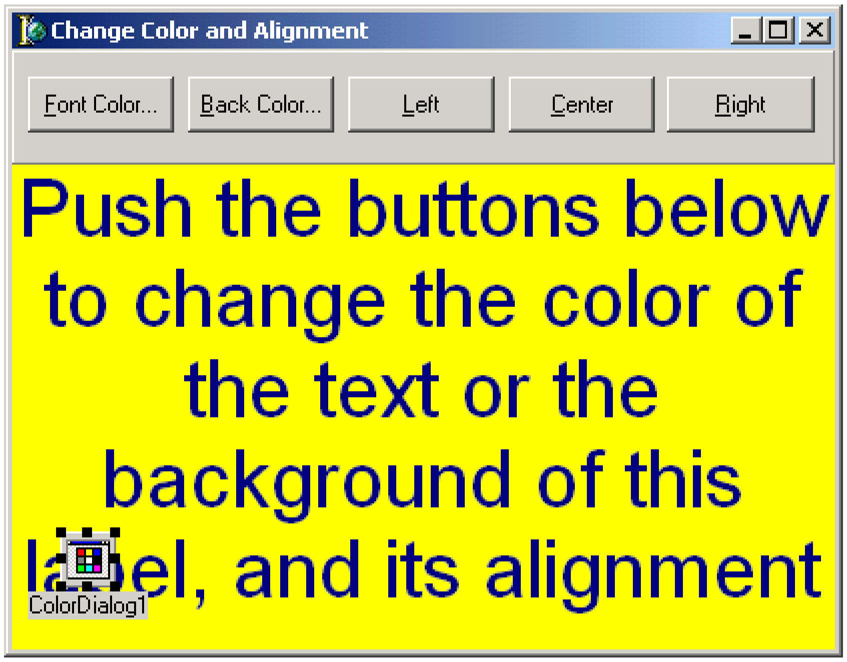
We use labels to provide descriptions of other components, particularly edit fields and list or combo boxes, because they have no title. If you open a dialog box in any Windows application, you'll probably see some text. These are *static controls* (in Windows terms) or *labels* (in Delphi terms).

Windows implements labels as windows of the static class. Delphi, instead, implements labels as non-windowed, graphical components. This is very important since it allows you to speed up form creation and save some Windows resources. However, Delphi also includes a new component that corresponds to the Windows label, the StaticText component. This component has similar properties and the same events, and Borland seems to have added it mainly for ActiveX support. We will use this component in the next example.

Besides using labels for descriptions, we can use instances of this component to improve and add some color to the user interface of our application. This is what we are going to do in the next example, LabelCo. The basic idea of this application is to test a couple of properties of the label component at run-time. Specifically, we want to alter the background color of the label, the color of its font, and the alignment of the text.

The LabelCo Example

The first thing to do is to place a big label in the form and enter some text. Write something long. I suggest you set the `WordWrap` property to `True`, to have several lines of text, and the `AutoSize` property to `False`, to allow the label to be resized freely. It might also be a good idea to select a large font, to choose a color for the font, and to select a color for the label itself.



Push the buttons below to change the color of the text or the background of this label, and its alignment

To change the font color, background color, and alignment properties of the label at run-time, we can use buttons. Instead of placing these buttons directly on the form, we can place a Panel component in the form, and then place the five buttons over (or actually inside) the panel. We need two to change the colors and three more to select the alignment — left, center, or right. Placing the buttons inside the panel, this last control will become the `Parent` component of the buttons: the coordinates of the buttons will be relative to the panel, so that moving the panel will move the buttons, hiding the Panel will hide the buttons, and so on.

The difference between the `Parent` and the `Owner` properties of a component is very important. The first indicates visual containment (like a button being inside a panel) while the latter indicates construction and destruction ownership (like a button being owned and destroyed by a form).

Making the label and the panel child components of the form allows us to align them (something you cannot do with buttons). Simply set the `Align` property of the Panel to `alTop`, and the `Align` property of the Label to `alClient`, and the two components will take up the full client area of the form. What's interesting is that when we resize the form, the two components will adjust their size and position correspondingly. Notice, by the way, that the StaticText component has no `Align` property.

The last component we have to place on the form is a ColorDialog component (you can find it in the Dialogs page of the Components palette). This component invokes the standard Windows Color dialog box. The resulting form is detailed in the following listing:

```
object ColorTextForm: TColorTextForm
  Caption = 'Change Color and Alignment'
  object Label1: TLabel
    Align = alClient
    Alignment = taCenter
    Caption = 'Push the buttons...' // omitted
    Color = clYellow
    Font.Color = clNavy
    Font.Height = -40
    Font.Name = 'Arial'
    WordWrap = True
  end
  object Panel1: TPanel
    Align = alTop
    object BtnFontColor: TButton...
    object BtnBackColor: TButton...
    object BtnLeft: TButton...
    object BtnCenter: TButton...
    object BtnRight: TButton...
  end
  object ColorDialog1: TColorDialog...
end
```

The two buttons used to change the color will display a dialog box, instead of performing an action directly. For this reason at the end of their `Caption` there is an ellipsis (...), which is the standard Windows convention for button and menu items to indicate the presence of a dialog box.

Now it's time to write some code. The click methods for the three alignment buttons are very simple. The program has to change the alignment of the label, as in:

```
procedure TColorTextForm.BtnLeftClick(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
end;
```

The other two methods should use the values `taCenter` and `taRightJustify` instead of `taLeftJustify`. You can find the names of these three choices in the `Alignment` property of the label, in the Object Inspector.

Writing code to change the color is a little more complex. In fact, we can provide a new value for the color, maybe choosing it from a list with a series of possible values. We might solve this problem, for example, by declaring an array of colors, entering a number of values, and then selecting a different element of the array each time. However, a more professional solution needs even less code: using the Windows standard dialog box to select a color.

The Standard Color Dialog Box

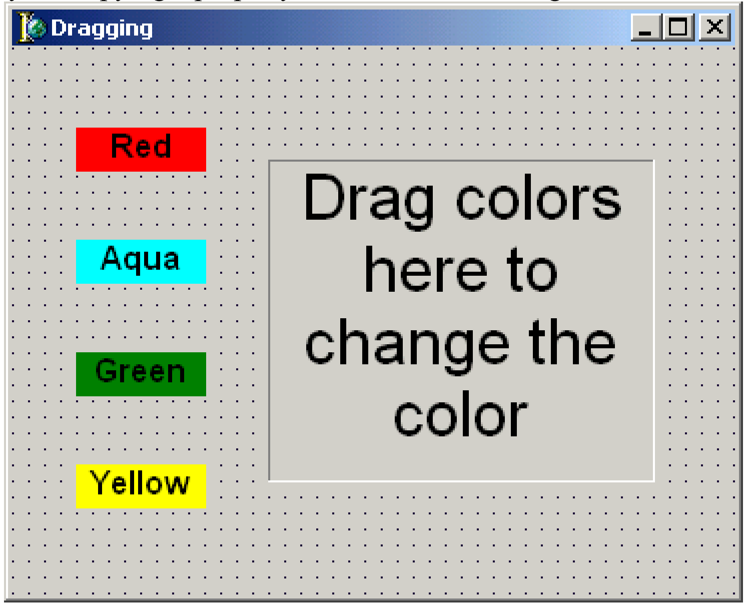
To use the standard Color dialog box, move to the Dialogs page of the Delphi Components palette, select the ColorDialog component, and place it anywhere on the form. The position has no effect, since at run-time this component is not visible inside the form. Now we can use the component, writing the following code:

```
procedure TColorTextForm.BtnFontColorClick(Sender: TObject);
begin
  ColorDialog1.Color := Label1.Font.Color;
  if ColorDialog1.Execute then
    Label1.Font.Color := ColorDialog1.Color;
end;
```

The three lines in the body of the procedure have the following meanings: with the first, we select the background color of the label as the initial color displayed by the dialog box; with the second, we run the dialog box; with the third, the color selected by the user in the dialog box is copied back to the label. We do this last operation only if the user closes the dialog box by pressing the OK button (in which case the `Execute` method returns `True`). If the user presses the Cancel button (and `Execute` returns `False`), we skip this last statement. To change the color of the label's text, we write similar code, referring this time to the `Label1.Font.Color` property. (The complete source code for the form LabelCo example is on the CD in the file `LABELF.PAS`.) You can see the Color dialog box in action in the figure below. This dialog box can also be expanded by clicking on the Define Custom Colors button.

Dragging from One Component to Another

Before we try out other Delphi components, it's helpful to examine a particular technique: *dragging.* The dragging operation is quite simple and is increasingly common in Windows. In Windows you can drag files and programs from a folder to another, drop them on the desktop, or perform similar dragging operations on files and folders with the Windows Explorer. You usually perform this operation by pressing the mouse button on one component (or window) and releasing it on another component (or window). When this operation occurs, you can provide some code, usually for copying a property, a value, or something else to the destination component.



As an example, consider the form in the figure above. There are four color labels, with the name of each color as text, and a destination label, with some descriptive text. Actually the destination label is implemented with a StaticText component. This component has a special value for the `Border` property, `sbsSunken`, with a *lowered* effect. A similar capability is not available for plain labels. The aim of this example, named Dragging, is to be able to drag the color from one of the labels on the left to the static text, changing its color accordingly. The components have very simple properties, as the following textual description of the form summarizes:

```
object DraggingForm: TDraggingForm
  Caption = 'Dragging'
  Font.Color = clBlack
  Font.Height = -16
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  object LabelRed: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Red'
    Color = clRed
    DragMode = dmAutomatic
  end
  object LabelAqua: TLabel...
  object LabelGreen: TLabel...
  object LabelYellow: TLabel...
  object StaticTarget: TStaticText
    Alignment = taCenter
    AutoSize = False
    BorderStyle = sbsSunken
    Caption = 'Drag colors here to change the color'
    Font.Height = -32
    OnDragDrop = StaticTargetDragDrop
    OnDragOver = StaticTargetDragOver
  end
end
```

After preparing the labels by supplying the proper values for the names and caption, as well as a corresponding color, you have to enable dragging. You can do this by selecting the value `dmAutomatic` for the `DragMode` property of the four labels on the left and responding to a couple of events in the destination label.

> As an alternative to the automatic dragging mode, you might choose the manual dragging mode. This is based on the use of the `BeginDrag` and `EndDrag` methods. An alternative approach is to handle dragging manually, simply by providing a handler for events related to moving the mouse and pressing and releasing mouse buttons.

The Code for the Dragging Example

The first event I want to consider is `OnDragOver`, which is called each time you are dragging and move the cursor over a component. This event indicates that the component accepts dragging. Usually, the event takes place after a determination of whether the `Source` component (the one that originated the dragging operation) is of a specific type:

```
procedure TDraggingForm.StaticTargetDragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TLabel;
end;
```

This code accepts the dragging operation, activating the corresponding cursor, only if the `Source` object is really a Label component. Notice the use of the `is` dynamic type checking operator. The second method we have to write corresponds to the `OnDragDrop` event:

```
procedure TDraggingForm.StaticTargetDragDrop(
  Sender, Source: TObject;  X, Y: Integer);
begin
  StaticTarget.Color := (Source as TLabel).Color;
end;
```

To read the value of the `Color` property from the `Source` object, we need to cast this object to the proper data type, in this case `TLabel`. We have to perform a type conversion — technically speaking, a type downcast (a typecast from a base class to a derived class, down through the hierarchy). A type downcast is not always safe. In fact, the idea behind this cast is that we receive the parameter `Source` of type `TObject`, which is really a label, and want to use it as a `TLabel` object, where `TLabel` is a class derived from`TObject`. However, in general, we face the risk of down-casting to `TLabel` an object that wasn't originally a label but, say, a button. When we start using the button as a label, we might have run-time errors.

In any case, when we use the `as` typecast, a type check is performed. Had the type of the `Source` object not been `TLabel`, an exception would have been raised. In this particular case, however, we haven't much to worry about. In fact, the `OnDragDrop` event is received only when the `Accept` parameter of the `OnDragOver` method is set to `True`, and we make this only if the `Source` object really is a `TLabel`.

Accepting Input from the User

We have seen a number of ways a user can interact with the application we write using a mouse: mouse clicks, mouse dragging, and so on. What about the keyboard? We know that the user can use the keyboard instead of the mouse to select a button by pressing the key corresponding to the underlined letter of the caption (if any).

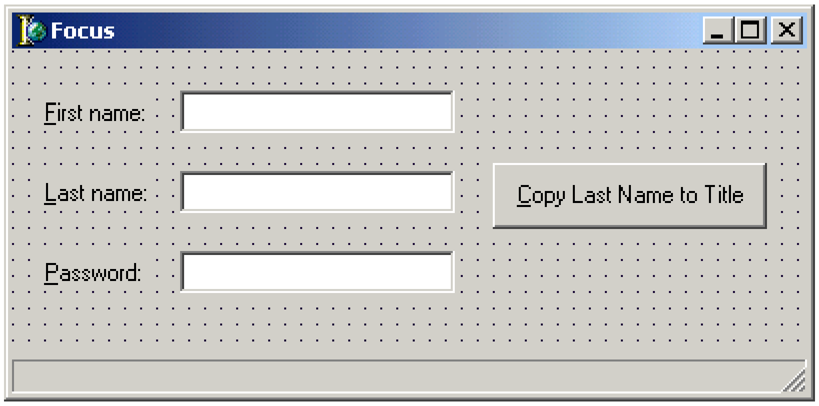
Aside from some particular cases, Windows can handle keyboard input directly. Defining handlers for keyboard-related events isn't a common operation, anyway. In fact, the system provides ready-to-use controls to build edit fields and a simple text editor. Delphi has several slightly different components in this area: Edit, MaskEdit, Memo, RichText, and the related data-aware controls. The two basic components are Edit and Memo.

An Edit component allows a single line of text and has some specific properties, such as one that allows only a limited number of characters or one that shows a special password character instead of the actual text. A Memo component, as we will see in a while, can host several lines of text.

Our first example of the Edit component, named Focus, will demonstrate a feature common to many controls, the *input focus*. In Windows, it's fairly simple to determine which is the active main window: it is in front of the other windows, and the title bar is a different color. It is not as easy to determine which window (or component) has the input focus. If the user presses a key, which component is going to receive the corresponding keyboard input message? It can be the active window, but it can also be one of its controls. Consider a form with several edit fields. Only one has the input focus at a given time. A user can move the input focus by using Tab or by clicking with the mouse on another component.

Handling the Input Focus

What's important for our example is that each time a component receives or loses the input focus, it receives a corresponding event indicating that the user either has reached (`OnEnter`) or has left (`OnExit`) the component. So we can add some methods to the form to take control over the input focus and display this information in a label or a status bar.



Besides three edit boxes, the form has also some labels indicating the meaning of the three edit fields (*First name*, *Last name*, and *Password*). For the output of the status information I've used a specific Windows common control, the StatusBar, but using a label or a panel would have had a similar effect. In fact, you can use the StatusBar component as a single-line output tool, by setting its `SimplePanel` property to `True`. Here is a summary of the properties for this example:

```
object FocusForm: TFocusForm
  Caption = 'Focus'
  object Label1: TLabel
    Caption = '&First name:'
    FocusControl = EditFirstName
  end
  object Label2: TLabel
    Caption = '&Last name:'
    FocusControl = EditLastName
  end
  object Label3: TLabel
    Caption = '&Password:'
    FocusControl = EditPassword
  end
  object EditFirstName: TEdit
    TabOrder = 0
    OnEnter = EditFirstNameEnter
  end
  object EditLastName: TEdit
    TabOrder = 1
    OnEnter = EditLastNameEnter
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    TabOrder = 2
    OnEnter = EditPasswordEnter
  end
  object ButtonCopy: TButton
    Caption = '&Copy Last Name to Title'
    TabOrder = 3
    OnClick = ButtonCopyClick
    OnEnter = ButtonCopyEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

As you can see, the form also contains a button we can use to copy the text of the `LastNameEdit` to the form's caption. This is just an example of how to work with text entered in an edit box. As you can see in the following code, before using the `Text` property of an edit box, it's a good idea to test whether the user has actually typed something or if the edit field is still empty:

```
procedure TFocusForm.ButtonCopyClick(Sender: TObject);
begin
  if EditLastName.Text <> '' then
    FocusForm.Caption := EditLastName.Text;
end;
```

Now we can move to the most interesting part of the program. We can write a comment in the status bar each time the focus is moved to a different control.

Displaying text in the status bar as the focus moves from control to control is a good way to guide the user through the steps of an application.

To accomplish this, we need four methods, one for each of the Edit components and one for the button, referring to the `OnEnter` event. Here is the code of one of the methods (the other three event handlers are very similar):

```
procedure TFocusForm.EditFirstNameEnter(Sender: TObject);
begin
  StatusBar1.SimpleText := 'Entering the first name...';
end;
```

You can test this program with the mouse or use the keyboard. If you press the Tab key, the input focus cycles among the Edit components and the button, without involving the labels. To have a proper sequence, you can change the `TabOrder` property of the windowed component. You can change this order either by entering a proper value for this property in the Object Inspector or (much better and easier) by using the Edit Tab Order dialog box, which can be called using the Tab Order command on the form's local menu. If you open this dialog box for the Focus example. Notice that the status bar is listed but you cannot actually move onto it using the Tab key.

A second way to select a component is to use a shortcut key. It is easy to place a shortcut key on the button, but how can you jump directly to an edit box? It isn't possible directly (the `Text` of the edit box changes as a user types), but there is an indirect way. You can add the shortcut key — the ampersand (`&`) — to a label, then set the `FocusControl` property of the label to the corresponding Edit component.

> Since Windows 95, the edit controls automatically have a local menu displayed when the user presses the right mouse button over them. Although you can easily customize such a menu in Delphi (as we will see in the next chapter), it is important to realize that this is standard behavior in the system.

A Generic OnEnter Event Handler

The problem with this code is that we have to write four different `OnEnter` event handlers, copying four strings to the text of the StatusBar component. To add more edit boxes to the example, you would need to add more event handlers, copying the code over and over. And if you wanted to provide a slightly different output (for example, by changing the output of the StatusBar allowing for multiple panels), you would need to change the code many times.

The alternative solution is to write a single event handler for the `OnEnter` event of each edit box (and the button, too). We simply need to store the message for the status bar in a property, then refer to this property for the `Sender` object. A good technique is to use the `Hint` property, which is actually designed for providing descriptions to the user.

Simply store the proper messages in the `Hint` property of the edit boxes and of the button, then remove the current `OnEnter` event handler, and install this method for each of them:

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
begin
  StatusBar1.SimpleText := (Sender as TControl).Hint;
end;
```

Notice you cannot write `Sender as TEdit` because the control might be a button as well. The solution is to typecast to a common ancestor class of `TButton` and `TEdit`, which defines the `Hint` property, as you can see in the code above.

Entering Numbers

We saw in the previous example that it is very easy to use an Edit component to ask the user to input some text, although it must be limited to a single line. In general, it's quite common to ask users for numeric input, too. To accomplish this, you can use the MaskEdit component (in the Additional page of the Components palette) or simply use an Edit component and then convert the input string into an integer, using the standard Pascal `Val` procedure or the Delphi `IntToStr` function.

This sounds good, but what if the user types a letter when a number is expected? Of course, these conversion functions return an error code, so we can use it to test whether the user has really entered a number. The second question is, when can we perform this test? Maybe when the value of the edit box changes, when the component loses focus, or when the user clicks on a particular button, such as the OK button in a dialog box. As you'll see, not all of these techniques work well.

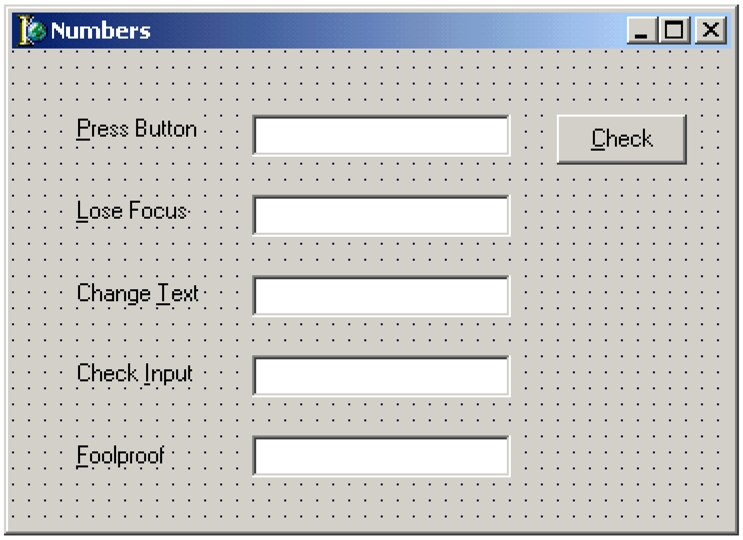
There is another, radically different, solution to the problem of allowing only numerical input in an edit box. You can look at the input stream to the edit box and stop any non-numerical input. This technique is not foolproof (a user can always paste some text into an edit box), but it works quite well and is easy to implement. Of course, you can improve it by combining it with one of the other techniques.

The next example, Numbers (see the form in the following figure), shows some of the techniques you can use to handle numerical input with an Edit component, so you can compare them easily. This example is meant as an exercise to discuss keyboard input and the input focus. To handle numerical input in an application you'll generally use specific components, as the SpinEdit or the UpDown controls available in Delphi. We will see an example of the use of another even more sophisticated control for keyboard input, the MaskEdit component.

In this example we're going to compare the effect of testing the input at different stages. First of all, build a form with five edit fields and five corresponding labels, describing in which occasion the corresponding Edit component checks the input. The form also has a button to check the contents of the first edit field. The contents of the first edit box are checked when the Check button is pressed. In the handler of the `OnClick` event of this button, the text is first converted into a number, using the `Val` procedure, which eventually returns an error code. Depending on the value of the code, a message is shown to the user:

```
procedure TNumbersForm.CheckButtonClick(Sender: TObject);
var
  Number, Code: Integer;
begin
  if Edit1.Text <> '' then
  begin
    Val (Edit1.Text, Number, Code);
    if Code <> 0 then
    begin
      Edit1.SetFocus;
      MessageDlg ('Not a number in the first edit',
        mtError, [mbOK], 0);
    end
    else
      MessageDlg ('OK, the number in the first edit box is' +
        IntToStr (Number), mtInformation, [mbOK], 0);
  end;
end;
```

If an error occurs, the application moves the focus back to the edit field before showing the error message to the user, thus inviting the user to correct the value. Of course, in this sample application a user can ignore this suggestion and move to another edit field.



The same kind of check is made on the second edit field when it loses the focus. In this case, the message is displayed automatically, but only if an error occurs. Why bother the user if everything is fine? The code here differs from that of the first edit field; it makes no reference to the `Edit2` component but always refers to the generic `Sender` control, making a safe typecast. To indicate the number of each button, I've used the `Tag` property, entering the number of the edit control.

As you can see in the following listing, instead of casting the `Sender` parameter to the `TEdit` class several times in the same method, it is better to do this operation once, saving the value in a local variable of the `TEdit` type. The type checking involved with these casts, in fact, is quite slow.

This method is a little more complex to write, but we will be able to use it again for a different component. Here is its code:

```
procedure TNumbersForm.Edit2Exit(Sender: TObject);
var
  Number, Code: Integer;
  CurrEdit: TEdit;
begin
  CurrEdit := Sender as TEdit;
  if CurrEdit.Text <> '' then
  begin
    Val (CurrEdit.Text, Number, Code);
    if Code <> 0 then
    begin
      CurrEdit.SetFocus;
      MessageDlg ('The edit field number ' +
        IntToStr (CurrEdit.Tag) + ' does not have a valid number',
        mtError, [mbOK], 0);
    end;
  end;
end;
```

Since Delphi 2, this code produces a warning message (a hint) when compiled, because the `Number` variable is not used after a value has been assigned to it. To avoid these hints, you can ask the compiler to disable its generation inside a specific method using the `$HINTS` compiler directive. Simply write `{$HINTS OFF}` before the method and `{$HINTS ON}` after it, as I've done in the source code of this example.

The third Edit component makes a similar test each time its content changes (using the `OnChange` event). Although we have checked the input on different occasions — using different events — the three functions are very similar to each other. The idea is to check the string once the user has entered it.

For the fourth Edit component, I want to show you a completely different technique. We are going to make a check *before* the Edit even knows that a key has been pressed. The Edit component has an event, `OnKeyPress`, that corresponds to the action of the user. We can provide a method for this event and test whether the character is a number or the Backspace key (which has a numerical value of 8, so we can refer to it as the character `#8`). If not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control, and produce a little warning sound:

```
procedure TNumbersForm.Edit4KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

The fourth Edit component accepts only numbers for input, but it is not foolproof. A user can copy some text to the Clipboard and paste it into this Edit control with the Shift+Ins key combination (but not using Ctrl+V), avoiding any check. To solve this problem, we might think of adding a check for a change to the contents, as in the third edit field, or a check on the contents when the user leaves the edit field, as in the second component. This is the reason for the fifth, Foolproof edit field: it uses the `OnKeyPress` event of the fourth edit field, the `OnChange` method of the third, and the `OnExit` event of the second, thus requiring no new code.

To reuse an existing method for a new event, just select the Events page of the Object Inspector, move to the component, and instead of double-clicking to the left of the event name, select the button in the combo box at the right. A list of names of old methods compatible with the current event — having the same number of parameters — will be displayed.

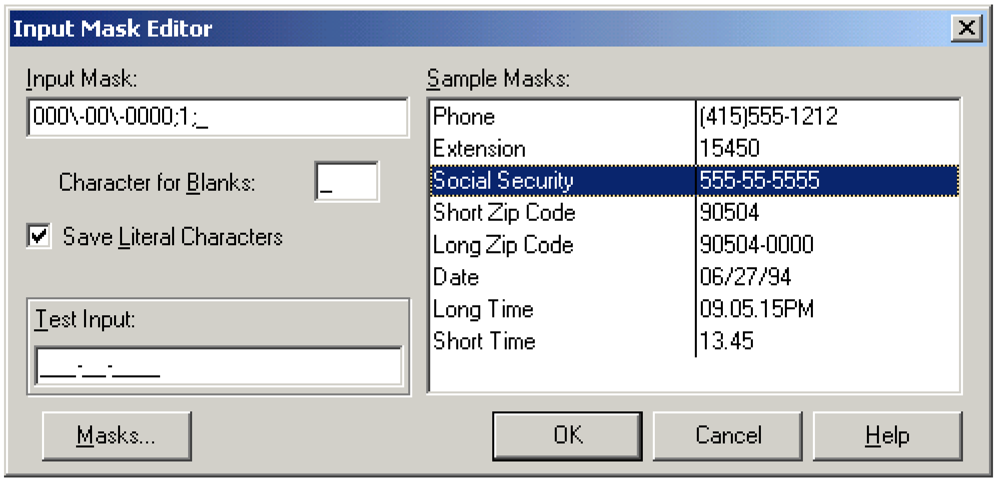
If you select the proper methods, the fifth component will combine the features of the third and the fourth. This is possible because in writing these methods, I took care to avoid any reference to the control to which they were related. The technique I used was to refer to the generic `Sender` parameter and cast it to the proper data type, which in this case was `TEdit`. As long as you connect a method of this kind to a component of the same kind, no problem should arise. Otherwise, you should make a number of type checks (using the `is` operator), which will probably make the code more complex to read. My suggestion is to share code only between controls of the same kind.

Notice also that to tell the user which edit box has incorrect text, I've added to each Edit component a value for the `Tag` property, as I mentioned before. Every edit box has a tag with its number, from 1 to 5.

Sophisticated Input Schemes

In the last example, we saw how an Edit component can be customized for special input purposes. The components could really accept only numbers, but handling complex input schemes with a similar approach is not straightforward. For this reason, Borland has supplied a ready-to-use *masked edit component*, an edit component with an input mask stored in a string.

For example, to handle numbers of no more than five digits, we can set the `EditMask` property to `99999`. (The character `9` stands for *non-compulsory digit*; refer to the Delphi documentation for the meaning of the various characters and symbols in the edit mask.) I suggest that you don't enter a string directly in this property, but instead always open the associated editor by clicking on the small ellipses button. The Input Mask editor has a test window and includes sample masks for commonly used input values.



Input Mask: 000\-00\-0000;1;_

Character for Blanks: _

[✓] Save Literal Characters

Test Input: ___-__-____

Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	06/27/94
Long Time	09:05:15PM
Short Time	13:45

Notice that the Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to have the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated, by default, with semicolons).

To see more default input masks, you can press the Masks button, which allows you to open a mask file. The predefined files hold standard codes grouped by country. For example, if you open the Italian group, you can find the taxpayer number (or *fiscal code*, which is used like social security numbers in the U.S.). This code is a complex mix of letters and numbers (including the consonants representing name, birth date, area code, and more), as its mask demonstrates:

```
LLLLLL00L00L000L
```

In this kind of code, `L` stands for a letter and `0` for a number. While you can look these up in the Help file, there is a summary of these codes in the following Mask1 example.

The form of this example includes a MaskEdit and an Edit component. The Edit is used to change the `EditMask` property of the first one at run-time. To accomplish this, I've just written a couple of lines of code to copy the text of the property into the edit box at the beginning (the `OnCreate` event) and reverse the action each time the plain edit box changes (`Edit1Change`):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := MaskEdit1.EditMask;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  MaskEdit1.EditMask := Edit1.Text;
end;
```

The form also has a list box with the description of the most important codes used to build the mask.

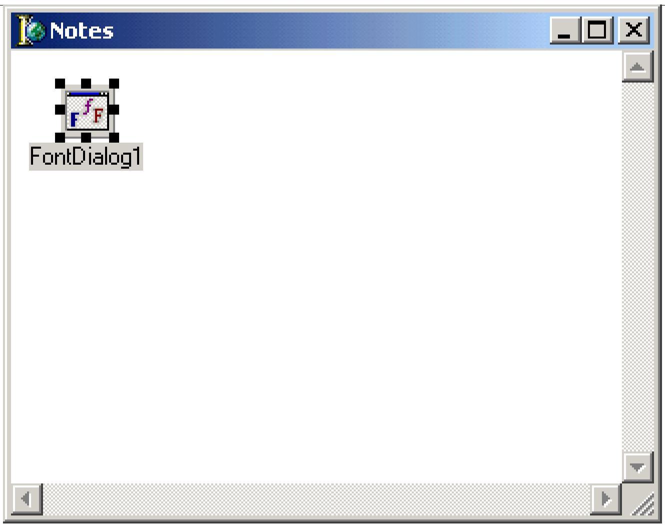
Creating a Simple Editor

Edit components can handle a limited amount of text, and only on a single line. If you need to accept longer text input, you should use the Memo component. A Memo component is like an Edit component, but it can span several lines, contain scroll bars to move through the text, and contain more text.

The easiest way to use a Memo is as a text editor, as you'll see in the next example, Notes. The idea is to implement an editor covering all of the window (or form) which contains it, to resemble Windows Notepad. The only other feature we will implement is to give the user the option of choosing the font for the editor.

Both parts are very easy to implement. Create a new project and place a Memo component on the form. Delete its text, remove the border, and set the `Alignment` property to `alClient`, so that it will always cover the whole client area — the internal surface — of the form. Also add both scroll bars, horizontal and vertical, selecting the value `ssBoth` for the memo's `ScrollBars` property. Here is the summary (and the design time image of the form):

```
object NotesForm: TNotesForm
  Caption = 'Notes'
  object Memo1: TMemo
    Align = alClient
    BorderStyle = bsNone
    Font.Height = -19
    Font.Name = 'Times New Roman'
    ScrollBars = ssBoth
    OnDblClick = Memo1DblClick
  end
  object FontDialog1: TFontDialog...
end
```



The Font Dialog Box

The second portion of the program involves the font. In the same way we used the standard Color dialog box in a previous example, we can use the standard Font selection dialog box provided by Windows. Just move to the Dialogs page of the Components palette and select the FontDialog component. Place it anywhere on the form, and add the following code when the user double-clicks inside the Memo:

```
procedure TNotesForm.Memo1DblClick(Sender: TObject);
begin
  FontDialog1.Font := Memo1.Font;
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```

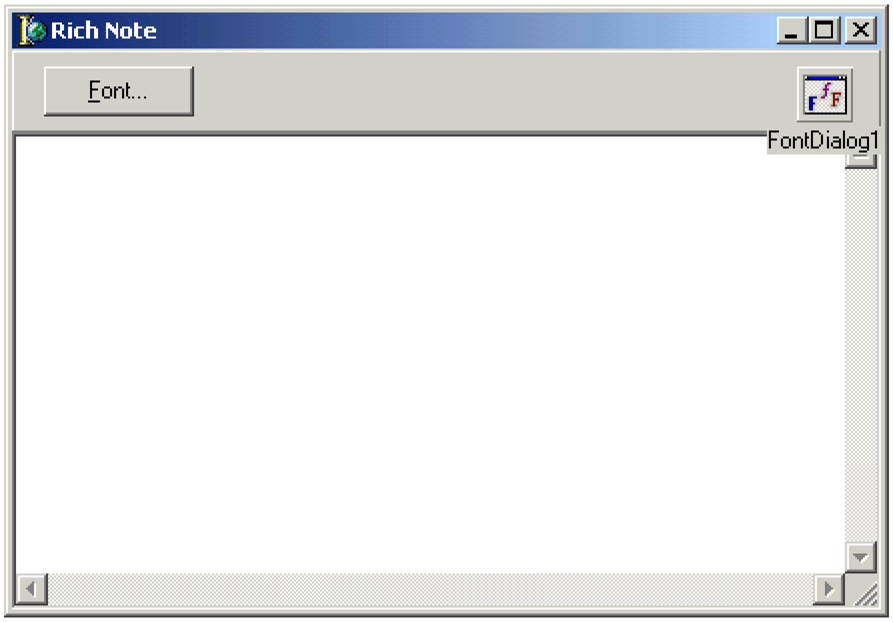
This code copies the current font to the corresponding property of the dialog component so it will be selected by default. Then it executes the dialog box. At the end, the `Font` property will contain the font the user selected. If the user presses the OK button, the third line of the above code copies the font back to the Memo.

This program is more powerful than it appears at first glance. For example, it allows copy and paste operations using the keyboard — this means you can copy text from your favorite word processor — and can handle the color of the font. Why not use it to place a big and colorful message on your screen?

Creating a Rich Editor

Although you can choose a font in the Notes program, all of the text you have written will have the same font. Windows has a control that can handle the Rich Text Format (RTF). A Delphi component, RichEdit, encapsulates the behavior of this standard control.

You can find an example of a complete editor based on the RichEdit component among the examples that ship with Delphi. (The example is named RichEdit, too). Here, we'll only change the previous program slightly by replacing the Memo component with a RichEdit, and allow a user to change the font of the selected portion of the text, not the whole text.



The RichNote example has a RichEdit component filling its client area. However, the component has no double-click event, so I added a button to select the font and placed it in a panel aligned to the top of the form, making a very simple toolbar. Here is the textual description of some of the properties of the three components:

```
object RichEdit1: TRichEdit
  Align = alClient
  HideScrollBars = False
  ScrollBars = ssBoth
end
object Panel1: TPanel
  Align = alTop
  object Button1: TButton
    Caption = '&Font...'
  end
end
```

Notice the caption of the button, which has an ampersand for the shortcut key, and an ellipsis at the end to indicate that pressing it will open a dialog box. When the user clicks on the button, if some text is selected, the program shows the standard Font dialog box using the default font of the RichEdit component as the initial value. At the end, the selected font is copied to the attributes of the current selection. The `DefAttributes` and `SelAttributes` properties of the RichEdit component are not of the `TFont` type, but they are compatible, so we can use the `Assign` method to copy the value:

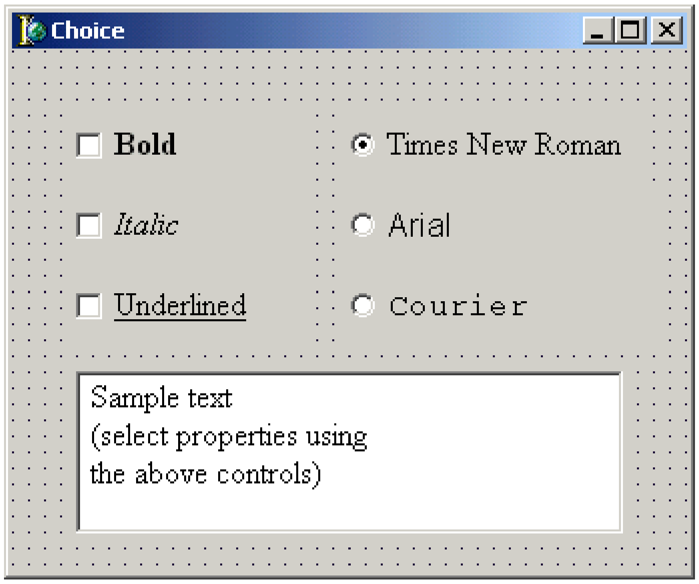
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign(RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  end
  else
    ShowMessage ('No text selected');
end;
```

The RichEdit component has other attributes related to fonts and paragraph formatting. We will use this component in further examples of the book; however, the simple code above is enough to let users produce much more complex output than the Memo component allows.

Making Choices

There are two standard Windows controls that allow the user to choose different options. The first is the *check box*, which corresponds to an option that can be selected freely (unless it has been disabled). The second control is the *radio button*, which corresponds to an exclusive selection. For example, if you see two radio buttons with the labels *A* and *B*, you can select *A* or select *B*, but not both of them at the same time. The other characteristic of a multiple choice is that you *must* check one of the radio buttons.

If the difference between check boxes and radio buttons is still not clear, an example might help you. The example has three check boxes to select the style *Bold*, *Italic*, or *Underlined*, and three radio buttons to choose a



Sample text
(select properties using
the above controls)

font (*Times New Roman*, *Arial*, or *Courier*). There is also a memo field with some text to show the effect of the user selections immediately.

The difference between the use of the check boxes and the radio buttons should be obvious. The text might be bold and italic at the same time, but it cannot be Arial and Courier at once. A user must choose only one font (and cannot choose none) but can select each of the styles independently from the other two (including no style at all).

This program requires some simple code. Each time the user clicks on a check box or radio button, we have to create a corresponding action. For the text styles, we have to look at the `Check` property of the control and add or remove the corresponding element from the memo's Font property `Style` set:

```
procedure TForm1.CheckBoldClick(Sender: TObject);
begin
  if CheckBold.Checked then
    Memo1.Font.Style := Memo1.Font.Style + [fsBold]
  else
```

```
    Memo1.Font.Style := Memo1.Font.Style - [fsBold];
end;
```

The other two check boxes have similar code for their `OnClick` events. The basic code for the radio buttons is even simpler since you cannot deselect a radio button by clicking on it:

```
procedure TForm1.RadioTimesClick(Sender: TObject);
begin
  Memo1.Font.Name := 'Times New Roman';
end;
```

Grouping Radio Buttons

Radio buttons represent exclusive choices. However, a form might contain several groups of radio buttons. Windows cannot determine by itself how the various radio buttons relate to each other. The solution, both in Windows and in Delphi, is to place the related radio buttons inside a container component. The standard Windows user interface uses a group box control to hold the radio buttons together, both functionally and visually. In Delphi, this control is implemented in the GroupBox component. However, Delphi has a second, similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio button inside it.

Using the radio group is probably easier than using the group box, but I'll use the more traditional approach to show you the code you can write to work with controls that have been placed inside another control. The RadioGroup component can automatically align its own internal radio buttons, and you can easily add new choices at run-time. You can see the differences between the two approaches in the next example.

The rules for building a group box with radio buttons are very simple. Place the GroupBox component in the form, then place the radio buttons in the group box. The GroupBox component contains other controls and is one of the container components used most often, together with the Panel component. If you disable or hide the group box, all the controls inside it will be disabled or hidden.

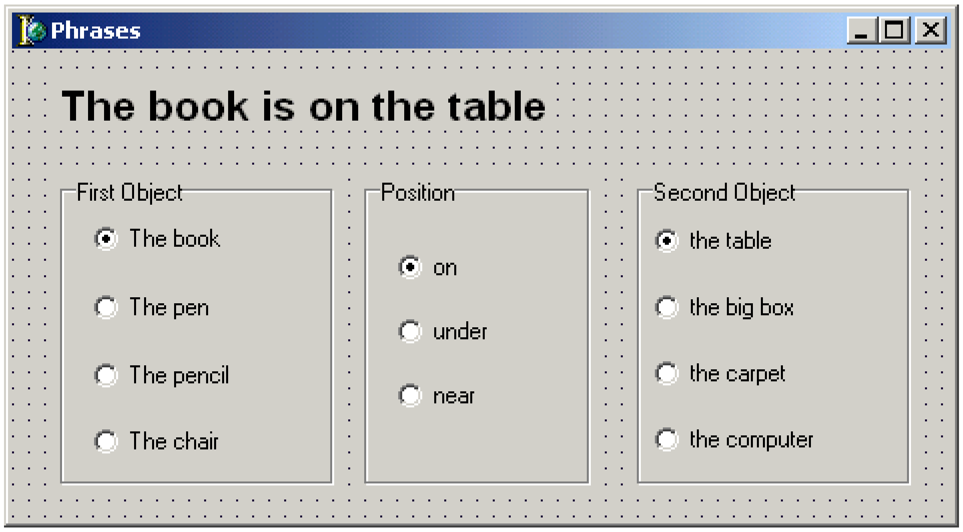
You can continue handling the individual radio buttons, but you might as well navigate through the array of controls owned by the group box. As discussed in the last chapter, the name of this property referring to this array of controls is `Controls`. Another property, `ControlCount`, holds the number of elements. These two properties can be accessed only at run-time.

The Phrases1 Example

If you've ever tried to learn a foreign language, you probably spent some time repeating the same silly and useless phrases over and over. Probably the most typical, when you learn English, is the infamous *"The book is on the table."* To demonstrate radio buttons, the Phrases1 example creates a tool to build such phrases by choosing among different available options.

This form is quite complex. If you rebuild it, remember that you must place the GroupBox components first and the radio buttons later. After doing this, you have to enter a proper caption for each element. The last selection is based on a radio group component, instead of a group box holding some radio buttons (as you can see in the textual description of the form below). In this case you create the options by entering a list of values in the Items property.

Remember that you also need to add a label, select a large font for it, and enter text corresponding to the radio buttons that are checked at design-time. This is an important point: When you place some radio buttons in a form or in a group box, remember to check one of the elements at design-time. One radio button in each group should always be checked, and the `ItemIndex` property of the radio group, indicating the current selection, should have a proper value.



Here is the textual description of the form with a summary of this information:

```
object Form1: TForm1
  Caption = 'Phrases'
  object Label1: TLabel
    Width = 243
    Caption = 'The book is on the table'
    Font.Height = -21
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object GroupBox1: TGroupBox
    Caption = 'First Object'
    object RadioBook: TRadioButton
      Caption = 'The book'
      Checked = True
      OnClick = ChangeText
    end
    object RadioPen: TRadioButton
      Caption = 'The pen'
      OnClick = ChangeText
    end
    object RadioPencil: TRadioButton...
    object RadioChair: TRadioButton...
  end
  object GroupBox2: TGroupBox
    Caption = 'Position'
    object RadioOn: TRadioButton
      Caption = 'on'
      Checked = True
      OnClick = ChangeText
    end
    object RadioUnder: TRadioButton...
    object RadioNear: TRadioButton...
  end
  object RadioGroup1: TRadioGroup
    Caption = 'Second Object'
    Items.Strings = (
      'the table'
      'the big box'
      'the carpet'
      'the computer')
    OnClick = ChangeText
  end
end
```

Now we have to write some code so that when the user clicks on the radio buttons, the phrase changes accordingly. There are different ways to do this. One is to follow the same approach as in the last example, providing a method for each button's `OnClick` event. Then we need to store the various portions of the phrase in some of the form's variables, change the portion corresponding to that button, and rebuild the whole phrase.

An alternative solution is to write a single method that looks at which buttons are currently checked and builds the corresponding phrase. This single method must be connected to the `OnClick` event of every radio button and of the RadioGroup component, a task we can easily accomplish. Select each of the radio buttons on the form (clicking on each one while you hold down the Shift key) and enter the name of the method in the Object Inspector. Since the method used to compute the new phrase doesn't refer to a specific control, you might name it yourself, simply entering a name in the second column of the Object Inspector next to the `OnClick` event. Here is the code of this single complex method:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: string;
  I: integer;
begin
  {look at which radio button is selected
  and add its text to the phrase}
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Phrase := (GroupBox1.Controls[I] as TRadioButton).Caption;

  {add the verb and blank spaces}
  Phrase := Phrase + ' is ';

  {repeat the operation on the second group box}
  for I := 0 to GroupBox2.ControlCount - 1 do
    with GroupBox2.Controls[I] as TRadioButton do
      if Checked then
        Phrase := Phrase + Caption;

  {retrieve the radio group selection, and display
  the result in the label}
  Label1.Caption := Phrase + ' ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex];
end;
```

The `ChangeText` method starts looking at which of the first group of radio buttons is selected, then moves on to adding a verb and the proper spaces between words. To determine which control in a group box is checked, the procedure scans these controls in a `for` loop. The `for` loop ranges from 0 to the number of controls minus 1, because the `Controls` array is zero-based, and tests whether the `Checked` property of the radio button is `True`. A cast is required to perform this operation — we cannot use the `Checked` property on a generic control. When the checked radio button has been found, the program simply copies its caption to the string. At this point, the `for` loop might terminate, but since only one radio button is checked at a time, it is safe to let it reach its natural end — testing all the elements. The same operation is repeated two times, but you can see that the second time a `with` statement is used to make the code shorter and more readable.

As you can see from the final portion of the method above, if you are using the RadioGroup component, the code is much simpler. This control, in fact, has an `ItemIndex` property indicating which radio button is selected and an `Items` property with a list of the text of the fake radio buttons. Overall, using a radio group is very similar to using a list box (as we will see in the next example), aside from the obvious difference in the user interface of the two components.

A List with Many Choices

If you want to add many selections, radio buttons are not appropriate, unless you create a really big form. The usual number of radio buttons is no more than 5 or 6. Another problem is that although you can disable a radio button, the elements of a group are usually fixed. Only when using a radio group can you have some flexibility. For both of these problems, the solution is to use a list box. A list box can host a large number of choices in a small space, because it can contain a scroll bar to show on screen only a limited portion of the whole list. Another advantage of a list box is that you can easily add new items to it or remove some of the current items. List boxes are extremely flexible and powerful.

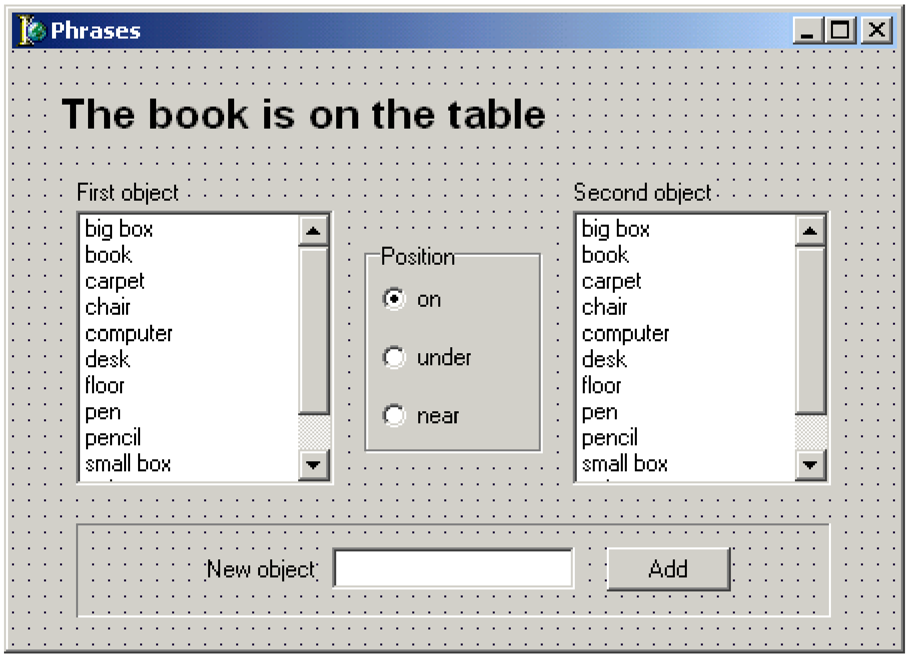
Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, a behavior similar to a group of radio buttons, and allowing multiple selections, which is similar to a group of check boxes. The next version of this example will have a multiple-selection list box.

For the moment, let's focus on a single-selection list box. We might use a couple of these components to change the Phrases1 example slightly. Instead of having a number of radio buttons to select the first and second objects of the phrase, we can use two list boxes. Besides allowing us to have a larger number of items, the advantage is that we can allow the user to insert new objects in the list and prevent selection of the same object twice, to avoid a phrase such as "The book is on the book." As you might imagine, this example is really much more complicated than the previous one and will require some fairly complex code.

The Form of the Phrases2 Example

As usual, the first step is to build a form. You can start with the form from the last example and remove the two group boxes on the sides and replace them with two list boxes. The radio buttons inside the group boxes will be deleted automatically. I've also replaced the central group box with a radio group. Actually, there's not much left from the previous example!

Now, add some strings to the `Items` property of both list boxes. For the example to work properly, the two list boxes should have the same strings; you can copy and paste them from the editor of the `Items` property of one list box to the editor of the same property of the other component. To improve the usability of the program, you might sort the strings in the list boxes, setting their `Sorted` property to `True`. Remember also to add a couple of labels above the list boxes, to describe their contents.



In the lower part of the form, I've also added an edit field, with its label, and a button, and a bevel around them to group them visually (the bevel is just a graphical component, not a container). As we will see later, when a user presses the button, the text in the Edit control is added to both list boxes. This operation will take place only if the text of the edit box is not empty and the string is not already present in the list boxes.

Here is the textual description of the components of this updated form (which is really very different from the previous version):

```
object Form1: TForm1
  Caption = 'Phrases'
  OnCreate = FormCreate
  object Label1: TLabel... // as in Phrases1
  object Label2: TLabel
    Caption = 'First object'
  end
  object Label3: TLabel
    Caption = 'Second object'
  end
  object ListBox1: TListBox
    Items.Strings = (
      'big box'
      'book'
      'carpet'
      'chair'
      'computer'...)
    Sorted = True
    OnClick = ChangeText
  end
  object RadioGroup1: TRadioGroup
    Caption = 'Position'
    Items.Strings = (
      'on'
      'under'
      'near')
  end
  object ListBox2: TListBox... // identical to ListBox1
  object Bevel1: TBevel...
  object Label4: TLabel
    Caption = 'New object'
  end
  object EditNew: TEdit...
  object ButtonAdd: TButton
    Caption = 'Add'
    OnClick = ButtonAddClick
  end
end
```

Working with the List Boxes

Once you have built this or a similar form, you can start writing some code. The first thing to do is to provide a new `ChangeText` procedure, connected with the `OnClick` event of the radio group and of the two list boxes. This procedure is simpler than in the previous example. In fact, to retrieve the selected text from the list box, you only need to get the number of the item selected (stored in the run-time property `ItemIndex`) and then retrieve the string at the corresponding position of the Item array, as the Phrases1 program did.

Here is what the code for the procedure looks like initially (this is just a temporary version, different from the one in the source code):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: String;
begin
  Phrase := 'The ';
  Phrase := Phrase + ListBox1.Items [ListBox1.ItemIndex];
  Phrase := Phrase + ' is ';
  Phrase := Phrase + RadioGroup1.Items [RadioGroup1.ItemIndex];
  Phrase := Phrase + ' the ';
  Phrase := Phrase + ListBox2.Items [ListBox2.ItemIndex];
  Label1.Caption := Phrase;
end;
```

This program, however, won't work properly because, at the beginning, no item is selected in either list box. To solve this problem, we can add some code to the form's `OnCreate` event. In this code, we can look for the two default strings, book and table, and select them. You should do this operation in two steps. First, you need to look for the string's index in the array of strings, with the `IndexOf` method. Then you can use that value as the index of the currently selected item:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
```

```
  N := ListBox1.Items.IndexOf ('book');
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf ('table');
  ListBox2.ItemIndex := N;
end;
```

Removing a Selected String from the Other List Box

Once this part of the program works, we have two more problems to solve: We must remove the selected string from the other list box (to avoid using the same term twice in a phrase), and we must write the code for the click event on the button.

The first problem is more complex, but I'll address it immediately since the solution of the second problem will be based partially on the code we write for the first one. Our aim is to delete from a list box the item currently selected in the other list box. This is easy to code. The problem is that once the selection changes, we have to restore the previous items, or our list boxes will rapidly become empty. A good solution is to store the two currently selected strings for the two list boxes in two private fields of the form, `String1` and `String2`:

```
type
  TForm1 = class(TForm)
  ...
 private
   String1, String2: String;
  end;
```

Now we have to change the code executed at startup and the code executed each time a new selection is made. In the `FormCreate` method, we need to store the initial value of the two strings and remove them from the other list box; the first string should be removed from the second list box, and vice versa. Since the `Delete` method of the `TStrings` class requires the index, we have to use the `IndexOf` function again to determine it:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
  String1 := 'book';
  String2 := 'table';

  {delete the selected string from the other list box
  to avoid a double selection}
  ListBox2.Items.Delete (ListBox2.Items.IndexOf (String1));
  ListBox1.Items.Delete (ListBox1.Items.IndexOf (String2));

  {select the two strings in their respective list boxes}
  N := ListBox1.Items.IndexOf (String1);
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf (String2);
  ListBox2.ItemIndex := N;
end;
```

The code to select the string should be executed after calling `Delete`, because removing an element before the one currently selected will alter the selection. The fact is that the selection is just a number referring to a string, not the reverse, as it should probably be. By the way, this doesn't depend on Delphi implementation but on the behavior of list boxes in Windows.

Things get complicated when a new item is selected in one of the list boxes. The `ChangeText` procedure has some new code at the beginning, executed only if the click took place on one of the list boxes (remember that the code is also associated with the group box). For each string, we have to check whether the selected item has changed and, in this case, add the previously selected string to the other list box and delete the new string. Here is the new version of the `ChangeText` method:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  TmpStr: String;
begin
  // if a list box has changed
  if Sender is TListBox then
  begin
    // get the text of the first string
    TmpStr := ListBox1.Items [ListBox1.ItemIndex];
    // if the first one has changed
    if TmpStr <> String1 then
    begin
      // update the strings in ListBox2
      {1.} ListBox2.Items.Add (String1);
      {2.} ListBox2.Items.Delete (
             ListBox2.Items.IndexOf (TmpStr));
      {3.} ListBox2.ItemIndex :=
             ListBox2.Items.IndexOf (String2);
      {4.} String1 := TmpStr;
    end;

    // get the text of the second string
    TmpStr := ListBox2.Items [ListBox2.ItemIndex];
    // if the second one has changed
    if TmpStr <> String2 then
    begin
      // update the strings in ListBox1
      ListBox1.Items.Add (String2);
      ListBox1.Items.Delete (ListBox1.Items.IndexOf (TmpStr));
      ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);
      String2 := TmpStr;
    end;
  end;

  // build the phrase with the current strings
  Label1.Caption := 'The ' + String1 + ' is ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex] +
    ' the ' + String2;
end;
```

What is the effect of the first part of this code? Here is a detailed description of the operations, referring to a new selection in the first list box. The procedure stores the selected element of the first list box in the temporary string `TmpStr`. If this is different from the older selection, `String1`, four operations take place (refer to the numbers in the listing above):

1. The previously selected string, `String1`, is added to the other list box, `ListBox2`.
2. The new selection, `TmpStr`, is removed from the other list box.
3. The selected string of the other list box, `String2`, is reselected in case its position has been changed by the two preceding operations.
4. Once the two lists contain the correct elements, we can store the new value in `String1` and use it later on to build the phrase.

We perform the same steps for the other list box a few lines later. Notice that we don't need to access the list boxes again to build the phrase at the end of the `OnChange` method, since `String1` and `String2` already contain the values we need.

Implementing the `OnClick` event for the Add button is quite simple. The only precautions we have to take are to test whether there is actually some text in the edit box or if it is empty and to check whether the string is already present in one of the two list boxes. Checking only one of the list boxes will miss a correspondence between the text of the edit box and the item currently selected in the other list box.

To make this check, we can ask both ListBox components for the index of the new string; if it is not present in the list — if there is no match — the value –1 will be returned. Otherwise, the `IndexOf` function returns the correct index, starting with 0 for the first element. In technical terms, we can say that the function returns the zero-based index of the element, or the error code –1 if it is not found. Here is the code:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  {if there is a string in the edit control and
  the string is not already present in one of the lists}
  if (EditNew.Text <> '') and
    (ListBox1.Items.IndexOf(EditNew.Text) < 0) and
    (ListBox2.Items.IndexOf(EditNew.Text) < 0) then
  begin
    {add the string to both list boxes}
    ListBox1.Items.Add (EditNew.Text);
    ListBox2.Items.Add (EditNew.Text);

    {reselects the current items properly}
    ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);
    ListBox2.ItemIndex := ListBox2.Items.IndexOf (String2);
  end
  else
    MessageDlg ('The edit control is empty or contains'
      + ' a string which is already present',
      mtError, [mbOK], 0);
end;
```

In the final part of this method's code, we need to reselect the current item of each list box since the position of the selected item might change. This happens if the new item is inserted before the one that is currently selected — that is, if it has a lower sort order.

Allowing Multiple Selections

A list box can allow the selection of either a single element or a number of elements. We make this choice in setting up a list box by specifying the value of its `Multiple` property. As the name implies, setting `Multiple` to `True` allows multiple selections. There are really two different kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case a user selects multiple items simply by clicking on them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items. This second choice is determined by the `ExtendedSelect` property.

While setting up a multiple-selection list box is very simple, the problems start to appear when you have to write the code. Accessing the selected item of a single-selection list box is simple. The `ItemIndex` property holds the index of the selected item, and the selected string can be retrieved with a simple expression:

```
ListBox2.Items[ListBox2.ItemIndex];
```

In a multiple-selection list box, on the other hand, we do not know how many items are selected, or even whether there is any item selected. In fact, a user can click on an item to select it, drag the cursor to select a number of consecutive items in the list, or click the mouse button on an item while holding down Ctrl key to toggle the selection of a single item without affecting the others. Using this last option, a user can even deselect all the items in a list box.

A program can retrieve information on the currently selected items by examining the `Selected` array. This array of `Boolean` values has the same number of entries as the list box. Each entry indicates whether the corresponding item is selected.

For example, to know how many items are selected in a list box, we need to scan the `Selected` array, usually with a `for` loop ranging from `0` to the number of items in the list minus one:

```
SelectCount := 0;
for ListItem := 0 to ListBox1.Items.Count - 1 do
  if ListBox1.Selected[ListItem] then
    Inc (SelectCount);
```

Actually, the ListBox component has an undocumented `SelCount` property, you can use to obtain exactly the information computed in the code above. We won't use this code in the next example, anyway, but a more complex version.

The Third Version of the Phrases Example

With this information, we can build a new version of the Phrases example, allowing a user to select several items in the first list box. The only real difference between the form of this new version and that of the last one is the value of the `MultiSelect` property in the first list box is not set to `True`.

In addition, the label at the top of the form has been enlarged, enabling the `WordWrap` property and disabling the `AutoSize` property, to accommodate longer phrases. Since the example's code is complex enough, I've removed the portion used to delete from a list box the item selected in the other list box. In the case of multiple selections, this would have been really complicated.

The main problem we face is building the different phrases correctly. The basic idea is to scan the `Selected` array each time and add each of the selected objects to the phrase. However, we need to place an *"and"* before the name of the last object, omitting the comma if there are only two. Moreover, we need to decide between singular and plural (*is* or *are*) and provide some default text if no element is selected. As you can see from the table below, building these phrases is not simple. In fact, if we store the phrase *"The book and the computer,"* when we need to add a third item, we must go back and change it.

Items Selected	**SelectCount**	**Phrase**
{none}	0	Nothing is
book	1	The book is
book, computer	2	The book and the computer are
book, computer, pen	3	The book, the computer, and the pen are
book, computer, pen, small box	4	The book, the computer, the pen, and the small box are

An alternative idea is to create two different phrases, one valid if no other elements will be added, the other prepared to host future objects (without the *and*). In the code, the `TmpStr1` string is the tentative final statement, while `TmpStr2` is the temporary string used to add a further element. At the end of the loop, `TmpStr1` holds the correct value. As you can see in the code below, in case only two items are selected we have to build the phrase in a slightly different way, removing the comma before the *and* conjunction.

Notice when scanning a sorted list box that the objects are always added to the resulting string in alphabetical order, not in the order in which they were selected. You can study how this idea has been implemented by looking at the new version of the `ChangeText` method, in the following code (and by studying the following table, which describes step-by-step how the strings are built):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase, TmpStr1, TmpStr2: String;
  SelectCount, ListItem: Integer;
begin
  SelectCount := 0;

  {look at each item of the multiple selection list box}
  for ListItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [ListItem] then
    begin
      {if the item is selected increase the count}
      Inc (SelectCount);
      if SelectCount = 1 then
      begin
        {store the string of the first selection}
        TmpStr1 := ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr1;
      end
      else if SelectCount = 2 then
      begin
        {add the string of the second selection}
        TmpStr1 := TmpStr1 + ' and the ' +
          ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr2 + ', the ' +
          ListBox1.Items.Strings [ListItem];
      end
      else // SelectCount > 2
      begin
```

```
        {add the string of the further selection}
        TmpStr1 := TmpStr2 + ', and the ' +
          ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr2 + ', the ' +
          ListBox1.Items.Strings [ListItem];
      end;
    end;

  {build the first part of the phrase}
  if SelectCount > 0 then
    Phrase := 'The ' + TmpStr1
  else
    Phrase := 'Nothing';

  if SelectCount <= 1 then
    Phrase := Phrase + ' is '
  else
    Phrase := Phrase + ' are ';

  {add the text of the radio button}
  Phrase := Phrase +
    RadioGroup1.Items [RadioGroup1.ItemIndex];

  {add the text of the second list box}
  Phrase := Phrase + ' the ' +
    ListBox2.Items [ListBox2.ItemIndex];
  Label1.Caption := Phrase;
end;
```

Items Selected	Steps	TmpStr1 (Tentative Final Statement)	TmpStr2 (Temporary Statement)
book	1	book	book
+ computer	2	book and the computer	book, the computer
+ pen	3	book, the computer, and the pen	book, the computer, the pen
+ small box	4	book, the computer, the pen, and the small box	book, the computer, the pen, the small box

The other procedures of the program change only slightly. The `FormCreate` method is simplified because we do not need to delete the selected item from the other list box. The `Add` method is simplified because both list boxes always have the same items and because the multiple-selection list box creates no problems with the selection if you add a new element.

An alternative solution to handle the status of multiple-selection list boxes is to look at the value of the `ItemIndex` property, which holds the number of the item of the list having the focus. If a user clicks on several items while holding down Ctrl, each time a click event takes place, you know which of the items have been selected or deselected — you can easily determine which of the two operations took place by looking at the value of the `Selected` array for that index. The problem is that if the user selects a number of elements by dragging the mouse, this method won't work. You need to intercept the dragging events, and this is considerably more complex than the technique described earlier.

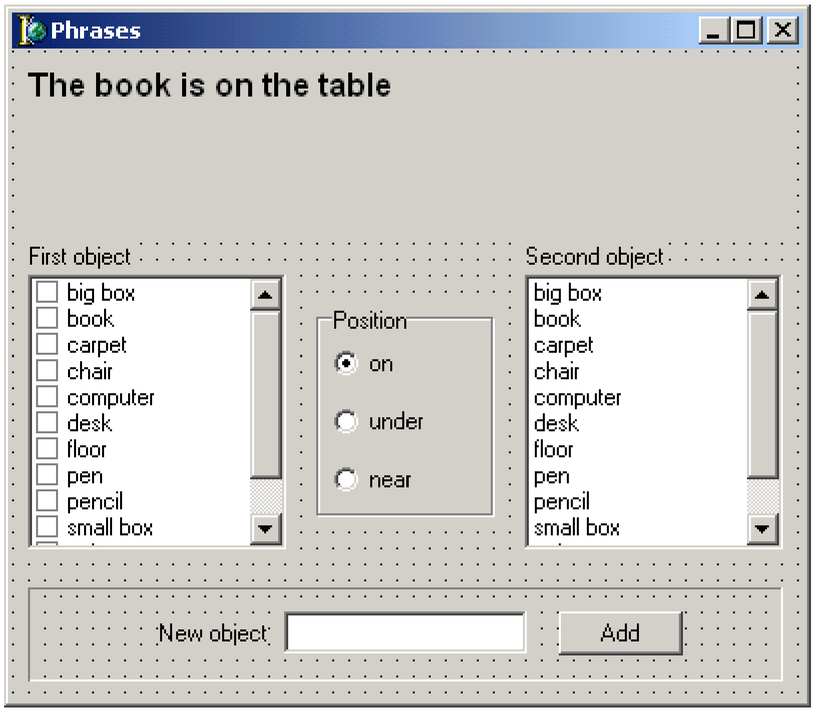
Using a CheckListBox Component

A further extension to the Phrases example is the use of the CheckListBox component, a component originally introduced by Borland in Delphi 3. This is basically a list box with a custom output (or an *owner-draw* list box, to use the proper technical term). Each item of the list is preceded by a check box. A user can select a single item of the list, but can also click on the check boxes to toggle their status.

If the component has the `AllowGrayed` property set to `True`, then each check box can be non-selected, grayed, or selected. Clicking on the check box alternates these three possible conditions. To check the current status of each item you can use the `Checked` and the `State` property. Both are array properties. The first, `Checked`, is a Boolean property you should use when `AllowGrayed` is set to `False`. The second, `State`, is a property of the `TCheckBoxState` data type:

```
type
  TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed)
```

This property should be used when `AllowGrayed` is set to `True`, to distinguish among the three different states of each item. Apart from these properties, the specific user interface, and the new `OnClickCheck` event, this component behaves as a ListBox.



As I mentioned at the beginning of this section, to show you an example of the use of this component I've further updated the Phrases3 example, building the Phrases4 version. I've basically replaced the first multiple selection list box with the new CheckListBox component, set its `Sorted` property to `True`, and copied the `Items`. Then I've updated the code, replacing the `ListBox1` object with the `CheckListBox1` object, and replacing the `Selected` property of the first with the `Checked` property of the second in the `ChangeText` method. Here is an excerpt of the new version of this method:

```
  for ListItem := 0 to CheckListBox1.Items.Count - 1 do
```

```
if CheckListBox1.Checked [ListItem] then
begin
  {if the item is selected increase the count}
  Inc (SelectCount);
  if SelectCount = 1 then
  begin
    {store the string of the first selection}
    TmpStr1 := CheckListBox1.Items.Strings [ListItem];
    TmpStr2 := TmpStr1;
  end
  else if SelectCount = 2 then
    ...
```

The important point of this program is that this component makes it more obvious to the user that the list box allows multiple selections. The plain list box, in fact, gives no clue of this fact.

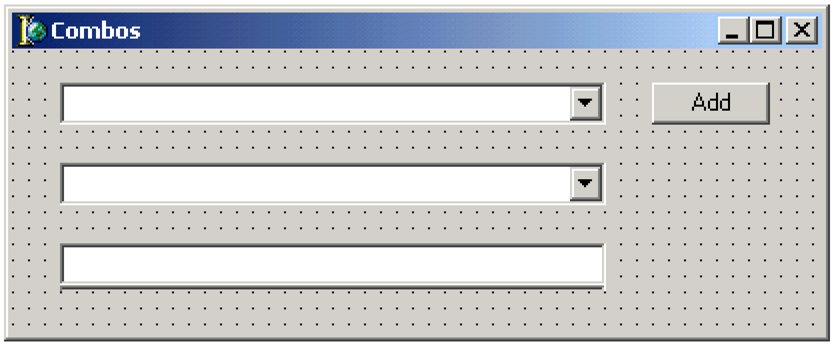
Many Lists, Little Space

List boxes take up a lot of screen space, and they offer a fixed selection. That is, a user can choose only among the items in the list box and cannot make any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control. A combo box is similar to an edit box, and you can often enter some text in it. It is also similar to a list box, with a drop-down arrow that displays a list box. Even the name of the control suggests that it is a combination of two other controls, an Edit and a ListBox. However, the behavior of a ComboBox component might change a lot, depending on the value of its `Style` property. Here is a short description of the various styles:

- The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request.
- The `csDropDownList` style defines a combo box that does not allow editing. By pressing a key, the user selects the first word starting with that letter in the list.
- The `csSimple` style defines a combo box that always displays the list box below it. This version of the control allows direct editing.
- The `csOwnerDrawFixed` and `csOwnerDrawVariable` styles define combo boxes based on an owner-draw list — that is, a list containing graphics determined by the program rather than simple strings.

To see the difference between the first three types, you can run the Combos example, which I'll describe in a moment. As you can appreciate by testing the program, Combos displays three combo boxes having three different styles: drop-down, drop-down list, and simple.



This program is very simple. Each combo box has the same basic strings—the names of more than 20 different animals. The first combo box contains an *Add* button. If the user presses the button, any text entered in the combo box is added to its list, provided it is not already present. This is the code associated with the `OnClick` event of the button:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
with ComboBox1 do
  if (Text <> '') and (Items.IndexOf (Text) < 0) then
    Items.Add (Text);
end;
```

You can use the second combo box to experiment with the automatic lookup technique. If you press a key, the first of the names in the list starting with that letter will be selected. By pressing the up arrow key and the down arrow key, you can further navigate in the list without opening it. This navigation technique of using initial letters and arrows can be used with each of the combo boxes.

The third combo box is a variation of the first. Instead of adding the new element when the Add button is pressed, that action is performed when the user presses the Enter key. To test for this event, we can write a method for the combo box's `OnKeyPress` event and check whether the key is the Enter key , which has the numeric code 13. The remaining statements are similar to those of the button's `OnClick` event:

```
procedure TForm1.ComboBox3KeyPress(
  Sender: TObject; var Key: Char);
begin
  {if the user presses the Enter key}
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

Delphi's `DateTimePicker` component has a user interface similar to that of a ComboBox.

Choosing a Value in a Range

The last basic component I want to explore in this chapter is the scroll bar. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. Notice, however, that when a scroll bar is associated with another component, it is really a portion of that component — one of its properties — and there is little relationship to the ScrollBar component itself. Forms having a scroll bar have no ScrollBar component. A portion of their border is used to display that graphical element.

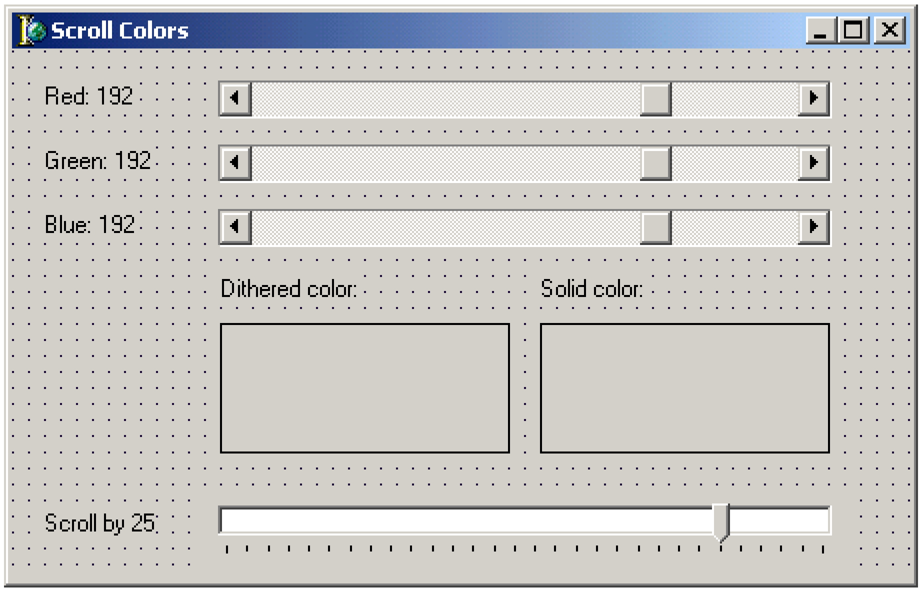
Direct usage of the ScrollBar component is quite rare, especially with the TrackBar Windows common control. However, there are cases in which it can play a role. The typical example is to allow a user to choose a numerical value in a large range (since a TrackBar is generally used for smaller ranges).

Most Windows programming books describe scroll bars using the example of selecting a color, and this book is no exception. But if you've seen a typical Windows example, you'll notice something very interesting: using Delphi, you can build this example in about one-fourth the time and writing a minimal amount of code.

The Scroll Color Example

The ScrollC example — the name stands for scroll color — has a simple form with three scroll bars and three corresponding labels, a track bar with its own label, and some shape components to show the current color. Each scroll bar refers to one of the three fundamental colors, which in Windows are red, green, and blue (RGB). Each label displays the name of the corresponding color and the current value.

Scroll bars have a number of peculiar properties. You can use `Min` and `Max` to determine the range of possible values; `Position` holds the current position; and the `LargeChange` and `SmallChange` properties indicate the increment caused by clicking on the bar or on the arrow at the end of the bar, respectively.



In the ScrollC example, the value of each bar ranges from `0` to `255`. The range is determined by the fact that each color is a `DWORD` with the lower three bytes representing the Red, Green, and Blue values (which is how colors are represented in Windows and in the VCL). The initial value of `192` has been chosen for the position because with settings of `192` for red, `192` for green, and `192` for blue, you get the typical light gray, which is the default value for the color of the form and of the shapes. Here is the textual description of one of these three ScrollBar components:

```
object ScrollBarRed: TScrollBar
  LargeChange = 25
  Max = 255
  Position = 192
  OnScroll = ScrollBarRedScroll
end
```

The TrackBar components has similar properties (`Min`, `Max`, `Position`):

```
object TrackBar1: TTrackBar
  Max = 30
  Min = 1
  Orientation = trHorizontal
  Frequency = 1
  Position = 25
  TickMarks = tmBottomRight
  TickStyle = tsAuto
  OnChange = TrackBar1Change
end
```

This control is used, in this example, to set the `LargeChange` property of the three scrollbars, with the following code:

```
procedure TFormScroll.TrackBar1Change(Sender: TObject);
begin
  LabelScroll.Caption := 'Scroll by ' +
    IntToStr(TrackBar1.Position);
  ScrollBarGreen.LargeChange := TrackBar1.Position;
  ScrollBarRed.LargeChange := TrackBar1.Position;
  ScrollBarBlue.LargeChange := TrackBar1.Position;
end;
```

When one of the scroll bars changes (the `OnScroll` event), the program has to update the corresponding label and the color of the shapes. The first of these shapes is used to show the color as it is determined by the three RGB values of the scroll bars. Assigning the color to the `Color` property of the brush used to fill the surface of the shape, we obtain a dithered color, an approximation of the real tint made with the colors available on the video adapter. The same color is assigned to the `Color` property of the pen of the second shape, resulting in the closest approximation of the requested color. Pens, in fact, do not use dithering, but rather the closest *pure* color. You can see the difference by running this example, although the effect might change depending on your video adapter.

If you browse through the code of the program, notice also that there is a third shape component used to mimic the border of the second shape. The real border of this shape, in fact, is enlarged to fill its whole surface, using a very wide pen. This way we use the color of the pen — the wide border — to actually fill the shape. Here is the code corresponding to one of the scroll bars:

```
procedure TFormScroll.ScrollBarRedScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  LabelRed.Caption := 'Red: ' + IntToStr(ScrollPos);
  Shape1.Brush.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
  Shape2.Pen.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
end;
```

You need to copy this code once for each scroll bar and correct the name of the label and its output text. The second and third statements always remain the same. They are based on a Windows function, `RGB`, which takes three values in the range 0–255 and creates a 32-bit value with the code of the corresponding color.

It is interesting to note that the `OnScroll` event has three parameters: the sender, the kind of event (`ScrollCode`), and the final position of the thumb (`ScrollPos`). This type of event can be used for very precise control of the user's actions. The `ScrollCode` parameter indicates if the user is dragging the thumb (`scTrack`, `scPosition`, or `scEndScroll`), has clicked on one of the two final arrows (`scLineUp` or `scLineDown`), has clicked on the bar in one of the two directions (`scPageUp` or `scPageDown`), or is trying to scroll out of the range (`scTop` or `scBottom`).

What's Next

In this chapter, we have started to explore some of the basic components available in Delphi. These components correspond to the standard Windows controls and some of the Windows common controls, and are extremely common in applications (with the exception of the stand-alone scroll bars). Of course, when you start adding more advanced Delphi components to an application, you can easily build more complex and colorful user interfaces and more powerful programs.

The next chapter is devoted to a specific and important topics: the use of menus. After that, we'll see more about forms, have a light excursus on multimedia, and wrap up the book with some simple specific techniques.

Chapter 5: Creating and Handling Menus

- The structure of a menu
- Using menu templates
- Checking, disabling, and modifying menus at run-time
- Creating menu items at run-time
- A custom menu check mark
- Bitmap menu items and owner-draw menu items
- The system menu
- Pop-up menus

The sample programs we have built so far have lacked one of the most important user-interface elements of any Windows application: the menu bar. Although our forms have each had a system menu, its use has been very limited. In practical applications, however, the menu bar is a central element in the development of a program. While the user can click and sometimes drag the mouse to select options, most complex tasks usually involve menu commands. Consider the applications you use and the number of menu commands you issue in those programs (including those invoked by a shortcut key, such as Ctrl+C, which is equivalent to the Edit | Copy command in most applications).

Menus are so important that almost any real Windows application has at least one. In fact, an application can also have several menus that change at run-time (more on this later), various local menus (usually activated with a right mouse click), and even a customized system menu.

The Borland programmers who created Delphi considered menus so important that they have placed the corresponding components in the Standard page of the Components palette.

The Structure of the Main Menu

Before looking at the use of menus in Delphi, let me recap some general information about menus and their structure. Usually, a menu has two levels. A menu bar, appearing below the title of the window, contains the names of the pull-down menus, each of which in turn contains a number of items. However, the menu structure is very flexible. It is possible to place a menu item directly in the menu bar and to place a *second level* pull-down menu inside another pull-down menu.

You should avoid placing commands directly on the menu bar, because users tend to select the elements of the menu bar to explore the structure of the menu. They do not expect to issue a command this way. If, for some reason, you really need to place a command in the menu bar, at least place the standard exclamation mark after it. Using an exclamation mark is a standard hint, but most users have never seen this “convention,” so it’s best to avoid the whole situation altogether and simply have a pull-down menu with a single menu item. A typical example is a Help menu with a single About menu item.

Putting a pull-down menu inside another pull-down menu — a second-level pull-down — is far more common, and Windows in this case provides a default visual clue, a small triangular glyph at the right of the menu. Many applications use this technique, because the system makes heavy use of multilevel menus (consider the Programs menu of the Start button). However, keep in mind that selecting a menu item in a second-level pull-down takes more time and can become tedious.

Many times, instead of having a second-level pull-down, you can simply group a number of options in the original pull-down and place two separator bars, one before and one after the group. You can see an exaggerated multilevel menu in the Levels example in the source code. Since this is a demonstration of what you should try to *avoid*, I won't list the structure of the menu here.

Different Roles of Menu Items

Now let's turn our attention to menu items, regardless of their position in the menu structure. There are three fundamental kinds of menu items:

- *Commands* are menu items used to execute an action. They have no special visual clue.
- *State-setters* are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active. In this case, selecting the command produces the opposite action.
- *Dialog menu items* are menu items that cause a dialog box to appear. The real difference between these and the other menu items is that a user should be able to explore the possible effects of the corresponding dialog box and eventually abort it by choosing the Cancel button. These commands should have a visual clue, consisting of an ellipsis (three dots) after the text.

> Besides the traditional state-setters with a check mark, you can also have radio menu items with a bullet check mark. These menu items represent alternative selections, just as RadioButton components do, and simply checking one of them disables the other elements of the group. We'll explore radio menu items in an example later on.

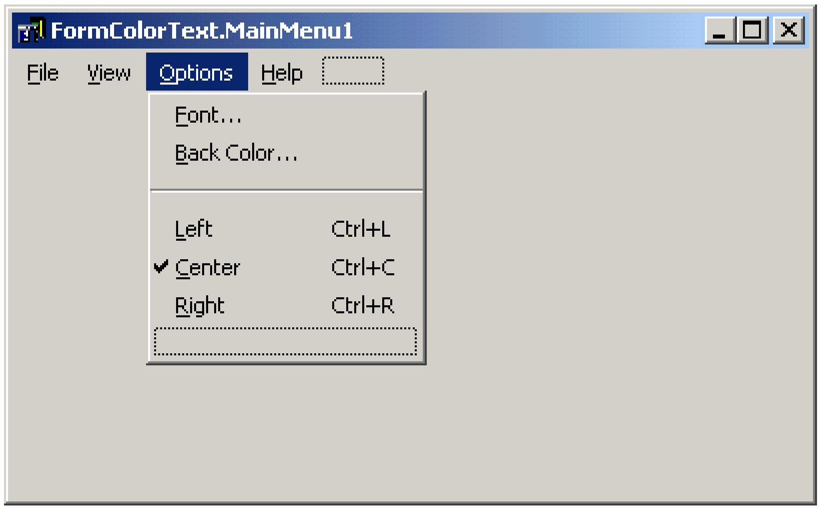
Building a Menu with the Menu Designer

Delphi includes a special editor for menus, the Menu Designer. To invoke this tool, place a MainMenu component on a form and double-click on it. Don't worry too much about the position of the menu component on the form, since it doesn't affect the result; the menu is always placed properly, below the form's caption.

> To be more precise, the form displays, below its caption, the menu indicated in its `Menu` property, which is set by default as soon as you create the first main menu component of the form. If the form has more than one main menu component, this property should be set manually and can be changed both at design-time and at run-time.

The Menu Designer is really powerful: It allows you to create a menu simply by writing the text of the commands, to move the items or pull-down menus by dragging them around, and to set the properties of the items easily. It is also very flexible, allowing you to place a command directly in the menu bar (this happens each time you do not write any element in the corresponding pull-down menu) or to create second-level pull-down menus.

To accomplish this, select the Create Submenu command on the Menu Designer's local menu (the local menu invoked with the right mouse button).



Another very important feature available through the Menu Designer is the ability to create a menu from a template. You can easily define new templates of your own. Simply create a menu, and use the Save As Template command on the local menu to add it to the list. This makes sense particularly if you need to have a similar menu in two applications or in two different forms of the same application.

The Standard Structure of a Menu

If you've used Windows applications for some time, you have certainly noticed that the structure of an application's menu is not an invention of its programmers. There are a number of standard Windows guidelines describing how to arrange the commands in a menu. You can infer most of these rules by looking at the menus of some of the best-selling applications.

An application's menu bar should start with a File pull-down, followed by Edit, View, and then some commands specific to the application. The final part of the sequence includes Options, Tools, and Window (in MDI, or Multiple Document Interface, applications) and always terminates with Help. Each of these pull-down menus has a standard layout, although the actual items depend on the application. The File menu, for example, usually has commands such as New, Open, Save, Save As, Print, Print Setup, and Exit.

Shortcut Keys and Hotkeys

A common feature of menu items is that they contain an underlined letter, generally called a *hotkey*. This letter, which is often the first letter of the text, can be used to select the menu using the keyboard. Pressing Alt plus the underlined key selects the corresponding pull-down menu. By pressing another underlined key on that menu, you issue a command.

Of course, each element of the menu bar must have a different underlined character. The same is true for the menu items on a specific pull-down menu. (Obviously, menu items on different pull-down menus can have the same underlined letter.) To indicate the underlined key, you simply place an ampersand (`&`) before it, as in `Save &As...` or `&File`. In these examples, the underlined keys would be `A` for `Save As` and `F` for `File`.

Menu items have another standard feature: shortcut keys. When you see the shorthand description of a key, or key combination, beside a menu item, it means you can press those keys to give that command. Although giving menu commands with the mouse is easier, it tends to be somewhat slow, particularly for keyboard-intensive applications, since you have to move one of your hands from the keyboard to the mouse. Pressing Alt and the underlined letter might be faster, but it still requires two operations. Using a shortcut key usually involves pressing a special key and another key at the same time (such as Ctrl+C). Windows doesn't even display the corresponding pull-down menu, so this results in a faster internal operation, too.

In Delphi, associating a shortcut key with a menu item (pull-down menus cannot have a shortcut key) is very easy. You simply select a value for the `ShortCut` property, choosing one of the standard combinations: Ctrl or Shift plus almost any key.

You might even add shortcut keys to a program without adding a real menu. For example, you can create a pop-up menu, connect it to a form (by setting the `PopupMenu` property of the form), set the `Visible` property of all of its items to `False`, and add the proper shortcut keys; a user will never see the menu, but the shortcuts (documented in your Help system, of course) will work. If this is not clear, you can look at the HShort example (the name stands for "Hidden Shortcut") in the book code.

Using the Predefined Menu Templates

To let you start developing an application's menu following the standard guidelines, Delphi contains some predefined menu templates. The templates include two different File pull-down menus, an Edit menu (including OLE commands), a Window menu, and two Help menus. There is also a complete MDI menu bar template, which has the same four menu categories.

Using these standard templates brings you some advantages. First of all, it is faster to reuse an existing menu than to build one from scratch. Second, the menu template follows the standard Windows guidelines for naming menu commands, for using the proper shortcuts, and so on. Of course, using these menus makes sense in a file-based application. But if the program you are writing doesn't handle files, has no editing capabilities, and is not MDI, you'll end up using only the template Help pull-down menu.

Responding to Menu Commands

To build the MenuOne example, the first example with a menu, we will extend the LabelCo example of the last chapter. The new version of the form has been extended with a MainMenu component. This menu bar has four pull-down menus: the File pull-down with only the Exit option, the View pull-down with only the Toolbar menu item, the Options menu with various options, and the Help menu with the About menu item.

To add the separator in the Options pull-down menu, simply insert a hyphen as the text of the command. Do not change the `Break` property. (Except for rare situations, the `Break` property will make a mess of your menu. You are better off forgetting that this property even exists.)

Of course, the `Break` property has its uses, or it would not have been added to the component. You can better understand it if I use different names for its possible values, NewLine or NewColumn. If an item on the menu bar has the `mbMenuBarBreak` (or NewLine) value, this item will be displayed in a second or subsequent line. If a menu item has the `mbMenuBreak` (or NewColumn) value, this item will be added to a second or subsequent column of the pull-down. Neither of these features are used very often.

The Code Generated by the Menu Designer

Once you have built this menu, take a look at the list of components displayed by the Object Inspector, or open the `DFM` file with the textual description of the form, which will also contain a textual description of the menu structure. Here is the portion of the textual description of the form related to the menu and its items:

```
object MainMenu1: TMainMenu
  object File1: TMenuItem
    Caption = '&File'
    object Exit1: TMenuItem
      Caption = 'E&xit'
      OnClick = Exit1Click
    end
  end
  object View1: TMenuItem
    Caption = '&View'
    object Toolbar1: TMenuItem
      Caption = '&Toolbar'
      Checked = True
      OnClick = Toolbar1Click
    end
  end
  object Options1: TMenuItem
    Caption = '&Options'
    object Font1: TMenuItem
      Caption = '&Font...'
      OnClick = Font1Click
    end
    object BackColor1: TMenuItem
      Caption = '&Back Color...'
      OnClick = BackColor1Click
    end
    object N1: TMenuItem
      Caption = '-'
    end
    object Left1: TMenuItem
      Caption = '&Left'
      ShortCut = 16460 // stands for Ctrl+L
      OnClick = Left1Click
    end
    object Center1: TMenuItem
      Caption = '&Center'
      Checked = True
      ShortCut = 16451 // stands for Ctrl+C
```

```
        OnClick = Center1Click
      end
      object Right1: TMenuItem
        Caption = '&Right'
        ShortCut = 16466 // stands for Ctrl+R
        OnClick = Right1Click
      end
    end
    object Help1: TMenuItem
      Caption = '&Help'
      object About1: TMenuItem
        Caption = '&About Menu One...'
        OnClick = About1Click
      end
    end
  end
```

As you can see in the listing above, there is a specific component for each menu item, one for each pull-down menu, and, surprisingly, even one for each separator. Delphi builds the names of these components automatically when you insert the menu item's label. The rules are simple:

- Any blank or special character (including ampersands and hyphens) is removed.
- If there are no characters left, the letter `N` is added.
- A number is always added at the end of the name (1 if this is the first menu item with this name, a higher number if not).

All of these new components are listed in the Object Inspector, and you can select them directly or navigate among them by opening the Menu Designer and selecting menu items visually. Actually each of these items is also listed as a component in the class definition of the form:

```
type
  TFormColorText = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    Font1: TMenuItem;
    BackColor1: TMenuItem;
    N1: TMenuItem;
    Left1: TMenuItem;
    Center1: TMenuItem;
    Right1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    File1: TMenuItem;
    Exit1: TMenuItem;
    View1: TMenuItem;
    Toolbar1: TMenuItem;
    ...
```

If there are menu items your code will not refer to (such as the separators), you can actually delete the fields declaring these objects (as `N1` above). They will be created anyway, since they are listed in the form definition file, but you won't be able to access them easily from within the source code of the form. Since the objects are created anyway, you won't save much memory, though (only the space for the reference inside the form class), but removing useless statements may improve code readability.

To respond to menu commands, you should define a method for the `OnClick` event of each menu item. The `OnClick` event of the pull-down menus is used only in special cases — for example, to check whether the menu items below should be disabled. The `OnClick` event of the separators is totally useless, because it will never be activated.

Once you have defined the main menu of a form and it is displayed below the caption, you can add a new method for the `OnClick` event of a menu command simply by selecting it in the menu bar. If a handler is already present, Delphi will show you the corresponding portion of the source code; otherwise, a new method will be added to the form.

The Code of the MenuOne Example

The code of the MenuOne example is very simple, and is similar to that of the LabelCo example it extends. The `OnClick` event of the menu command for the background color and of the corresponding toolbar button are connected to the same method:

```
object BtnBackColor: TButton
  Caption = '&Back Color...'
  OnClick = BackColor1Click
end
```

This method has the same code as the earlier version. The menu command and button related to the font are both connected to the `Font1Click` method, which this time displays the font selection dialog box, not the color selection dialog box. Notice that the code is quite compact, because it uses a `with` statement:

```
procedure TFormColorText.Font1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    Font := Label1.Font;
    if Execute then
      Label1.Font := Font;
  end;
end;
```

The other three menu items of the Options pull-down menu (and the last three buttons of the toolbar) have basically the same code as the LabelCo example: the code of their `OnClick` event handlers simply set the `Alignment` property of the label. This works fine, but the resulting application doesn't follow the standard user-interface guidelines. Each time you have a series of choices in a menu, the selected choice should have a check mark beside it.

To accomplish this, you need to create two different operations. First, you have to place a check mark near the default choice, Center, changing the value of the menu item's `Check` property in the Object Inspector. Second, you should correct the code so that each time the selection changes, the check mark is properly set:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Center1.Checked := False;
  Right1.Checked := False;
end;
```

The other two methods are similar. You can simply copy the source code of the last three statements, paste this text twice into the other two methods, and correct the values of the three `Checked` properties so that each time one of them is set to `True`. Removing the check marks from the other items of the group can be handled in more efficient ways, but the real solution is to use radio menu items, instead of check marks. We'll look at this technique later on.

The View | Toolbar menu item is a typical item with a check mark, set when the toolbar is visible. Here is its code:

```
procedure TFormColorText.Toolbar1Click(Sender: TObject);
begin
  Panel1.Visible := not Panel1.Visible;
  Toolbar1.Checked := Panel1.Visible;
end;
```

The program toggles the status of the `Visible` property of the panel, then sets the check mark — the `Checked` property — of the menu item accordingly. You should only remember to set the initial value of this property, to match the initial status of the panel. The Help | About and File | Exit commands simply show a message box or call the `Close` method of the form, respectively. They are so simple I won't show their source code here. (You can find it in the `MenuOneF.PAS` file among th source code.)

Modifying the Menu at Run-Time

You can perform a number of operations in Windows to change the structure of a menu at run-time. We'll start by examining in detail the operations you can do on a single menu item, then move to pull-down menus, build a flexible main menu, and change the default bitmap for the check mark.

Changing Menu Items at Run-Time

It is important to note that menu items can change at run-time. For example, when a menu command cannot or should not be selected, it is usually grayed. In this case, the user has no way to issue that command. You shouldn't generally hide a menu item when it is not available, but simply disable it. This standard technique lets users know that the command is currently not available. Otherwise, they might think the menu command was somewhere else in the menu structure, and keep looking for it.

Another visual change is the use of the check mark, which applications can toggle on and off easily, as we've seen in the last example. At times, to implement a state-setter menu item, you can change the text of the menu item altogether, which might result in an easier interface. For example, suppose an application has a Show Toolbar command. If you select it, a toolbar will appear and a check mark will be added to the item. This means that if you again select the Show Toolbar command, the toolbar will disappear: The command you issue has the opposite effect as its name. To avoid this problem, you might use two different captions for the two states of the menu item, such as *Show Toolbar* and *Hide Toolbar*.

When the user can select more than two choices, it is better to use multiple menu items with a check mark, or even better the new radio menu item user interface, which is becoming the standard approach.

Three properties are commonly used to modify a menu item:

- We used the `Checked` property in the example above to add or remove a check mark beside the menu item.
- The `Enabled` property can be used to gray a menu item so that it cannot be selected by a user (but it remains visible).
- The last property of this group is the `Caption`, the text of the menu item, which can be modified to reflect the actual effect of a command, as discussed above.

I'll demonstrate the use of these properties by extending the MenuOne example (the name of the new project is MenuOne2) with new menu items. I've added two new menu items to the View pull-down menu (Hide Label and Fixed Font) and two new menu items in the Options pull-down menu (Fixed View and Disable Help), plus a couple of new separators.

The Hide Label menu item demonstrates the use of different captions for an item depending on the status of the program. Here is its `OnClick` event handler:

```
procedure TFormColorText.HideLabel1Click(Sender: TObject);
begin
  Label1.Visible := not Label1.Visible;
  if Label1.Visible then
    HideLabel1.Caption := 'Hide &Label'
  else
    HideLabel1.Caption := 'Show &Label'
end;
```

Disabling Menu Items and Hiding Pull-Down Menus

The other three new menu items of the MenuOne2 example are used to disable or hide other menu items or pull-down menus. The View | Fixed Font command is used to disable the Options | Font menu item:

```
procedure TFormColorText.FixedFont1Click(Sender: TObject);
begin
  ToggleCheck (FixedFont1);
  Font1.Enabled := not Font1.Enabled;
end;
```

This method calls the custom `ToggleCheck` procedure I've written as a shortcut to the code for toggling the check mark (something all these three final menu commands do). Here is its simple code:

```
procedure ToggleCheck (Item: TMenuItem);
begin
  Item.Checked := not Item.Checked;
end;
```

I've already advised you against hiding menu items (by turning off their `Visible` property), because users will probably try to find them in a different pull-down menu. Menu items should generally be disabled when you want to prevent users from calling them. Hiding entire pull-down menus, however, is a common practice. In many applications the pull-down menus you see reflect the window you are working on. Technically, it is possible to disable a pull-down menu as well, but this is not very common. The last two menu items I've added to the program do these two operations, as you can see in their corresponding `OnClick` event handlers:

```
procedure TFormColorText.FixedView1Click(Sender: TObject);
begin
  ToggleCheck (FixedView1);
  View1.Visible := not View1.Visible;
end;

procedure TFormColorText.DisableHelp1Click(Sender: TObject);
begin
  ToggleCheck (DisableHelp1);
  Help1.Enabled := not Help1.Enabled;
end;
```

The View pull-down menu has been removed, and the Help pull-down menu is grayed.

Using Radio Menu Items

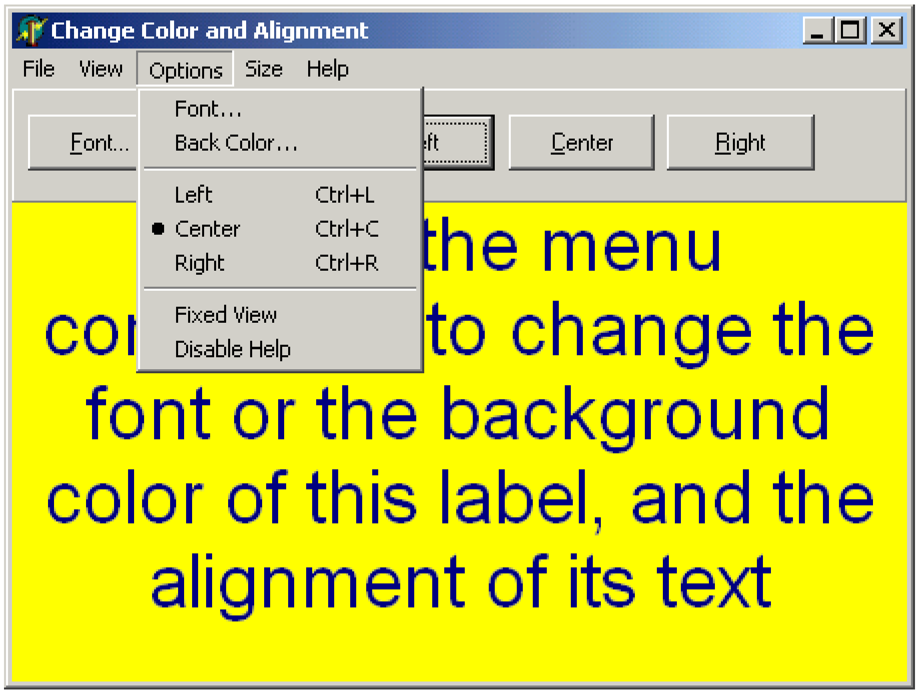
In addition to using check marks, in Windows you can use radio menu items. These provide not only a different user interface, but also different behavior (basically simpler code, since the system does some of the work for us). A notable example of this new user-interface feature is the View menu of the Windows Explorer.

In Delphi, simply set the `RadioItem` property of a MenuItem component to `True` and you get the new check mark for the item. If you set this property for several consecutive menu items and set their `GroupIndex` property to the same value, they'll use the new mark and behave as radio buttons. This means that only one of the menu items in the group will be selected at a time. Instead of having to deselect all other items manually, as you did in the first version of the MenuOne example, now you can simply select the proper menu item, and the rest is automatic.

Here is how you can implement this feature, in the third version of the MenuOne example (which also has another feature I'll discuss in the next section). The following listing shows the updated textual description of this group of menu items:

```
object Options1: TMenuItem
  Caption = '&Options'
  ...
  object Left1: TMenuItem
    Caption = '&Left'
    GroupIndex = 1
    RadioItem = True
    OnClick = Left1Click
  end
  object Center1: TMenuItem
    Caption = '&Center'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = Center1Click
  end
```

```
object Right1: TMenuItem
  Caption = '&Right'
  GroupIndex = 1
  RadioItem = True
  OnClick = Right1Click
end
```



This code will work as is, but we can actually simplify it. Here is the new version of the `Right1Click` method:

```
procedure TFormColorText.Right1Click(Sender: TObject);
begin
  Label1.Alignment := taRightJustify;
  // Left1.Checked := False; // now useless
  // Center1.Checked := False; // now useless
  Right1.Checked := True;
end;
```

I've simply commented out the two useless statements, instead of deleting them, to let you see the differences from the older version.

Creating Menu Items Dynamically

The run-time changes on menu items and pull-down menus we've seen so far were all based on the direct manipulation of some properties. These components, however, also have some interesting methods, such as `Insert` and `Remove`, that you can use to make further changes.

The basic idea is that each object of the `TMenuItem` class — which Delphi uses for both menu items and pull-down menus — contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on.

The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items (or entire pull-down menus) to a menu is fairly easy. Slightly more complex is the handling of the commands related to the new menu items. Basically, you need to write a specific message-response method in your code (without any help from the Delphi environment), and then assign it to the new menu item by setting its `OnClick` property. As an alternative, you can have a single method used for several `OnClick` events and use its `Sender` parameter to determine which menu command the user issued.

All these features are demonstrated by the MenuOne3 example. As soon as you start this program, it creates a new pull-down with menu items used to change the size of the font of the big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you. I could have created the pull-down at design-time, and then added the menu items dynamically, but I prefer showing you the complete code, so that you can apply it to other cases.

To create a new menu item (or pull-down) you simply call the `Create` constructor of the `TMenuItem` class:

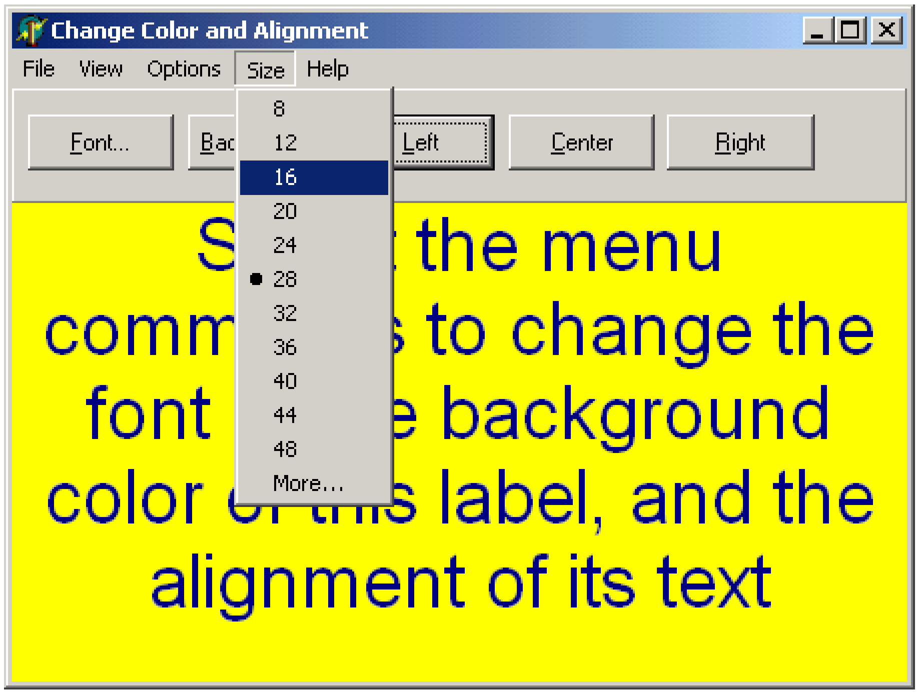
```
var
  PullDown: TMenuItem;
begin
  PullDown := TMenuItem.Create (self);
```

Then you can simply set its `Caption` and other properties, and finally insert it in the proper parent menu. The new pull-down should be inserted in `Items` of the `MainMenu1` component. You can calculate the position, knowing the index is zero-based, or you can ask the main menu component for the previous pull-down menu:

```
Position := MainMenu1.Items.IndexOf (Options1);
MainMenu1.Items.Insert (Position + 1, PullDown);
```

The menu items of this pull-down are created in a `while` loop (I don't use a `for` loop because I want to provide only one menu item for every four possible sizes: 8, 12, 16, and so on). The code to create each item is slightly more complex simply because I want to turn them into radio items, but the basic structure of the code is very simple:

```
var
  Item: TMenuItem;
  I: Integer;
begin
  ...
  I := 8;
  while I <= 48 do
  begin
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
```


Change Color and Alignment
File View Options Size Help
Font... Bac Left Center Right
8
12
16
20
24
● 28
32
36
40
44
48
More...
S the menu comm s to change the font e background color o s label, and the alignment of its text

To insert an item at the end I call the Insert method passing the number of items (`PullDown.Count`) as a parameter. As you can see, the program adds one extra item at the end of the menu, used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method, which shows the font selection dialog box:

```
Item := TMenuItem.Create (self);
Item.Caption := 'More...';
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
```

Also the `OnClick` events of the other menu items are connected with a method, but this time it is a method you have to define manually, by adding its declaration to the form class:

```
type
  TFormColorText = class(TForm)
    ...
  public
    procedure SizeItemClick(Sender: TObject);
end;
```

The method should have the proper signature (or parameters list). Here is the code of the method, which is based on the `Sender` parameter:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

As you can see, this code doesn't set the proper check mark (or radio item mark) next to the selected item. The reason is that the user can select a new size by changing the font. For this reason, we can use a different approach, and handle the `OnClick` event of the pull-down menu. This event is activated just before showing the pull-down menu, so we can use it to set the proper check mark at that time:

```
procedure TFormColorText.SizeClick (Sender: TObject);
```

```
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

This code scans the items of the pull-down menu we have activated (the `Sender`), skipping only the final one, and checks whether the caption matches the current `Size` of the font of the label. If no match is found, the program checks the last menu item, to indicate that a different size is active.

Of course we have to set this `SizeClick` event handler at run-time, when the pull-down menu is created. So we can finally look at the complete source code of the `FormCreate` method, which sums up all the features I have discussed in this section:

```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pulldown menu
  PullDown := TMenuItem.Create (self);
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);

  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;

  // add extra item at the end
  Item := TMenuItem.Create (self);
  Item.Caption := 'More...';
  // make it a radio item
  Item.GroupIndex := 1;
  Item.RadioItem := True;
  // handle click by showing the font dialog box
  Item.OnClick := Font1Click;
  PullDown.Insert (PullDown.Count, Item);
end;
```

Creating Menus and Menu Items Dynamically

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the MenuOne3 example. As an alternative, you can also use some global functions available in the Menus unit:

```
function NewMenu(Owner: TComponent; const AName: string;
  Items: array of TMenuItem): TMainMenu;
function NewPopupMenu(Owner: TComponent;
  const AName: string; Alignment: TPopupAlignment;
  AutoPopup: Boolean; Items: array of TMenuitem):
  TPopupMenu;
function NewSubMenu(const ACaption: string;
  hCtx: Word; const AName: string;
  Items: array of TMenuItem): TMenuItem;
function NewItem(const ACaption: string;
  AShortCut: TShortCut; AChecked, AEnabled: Boolean;
  AOnClick: TNotifyEvent; hCtx: Word;
  const AName: string): TMenuItem;
function NewLine: TMenuItem;
```

The `NewMenu` and `NewPopupMenu` functions, in particular, should be used to create brand-new menus. Calling the constructors of the corresponding classes, in fact, doesn't always work properly.

Short and Long Menus

If you don't like creating menu items dynamically, but still need to have a very flexible menu, there are a couple of good alternatives. You can create a large menu with all the items you need, then hide all the items and pull-down menus you do not want at the beginning. To add a new command you need only show it. This solution is a follow-up to what we have done up to now.

You can also create several menus, possibly with common elements, and exchange them as required. This approach is demonstrated in this section. A typical example of a form having two menus is one that uses two different sets of menus (long and short) for two different kinds of users (expert and inexperienced). This technique was common in major Windows applications for some years but has since been replaced by other approaches, such as letting each user redefine the whole structure of the menu.

The idea is simple and its implementation straightforward:

1. Prepare the full menu of the application, adding a menu item with the `Caption` *'Short'*.
2. Add this menu to the Delphi menu template.
3. Place a second MainMenu component on the form, and copy its structure from the template.
4. In the second menu, remove the items corresponding to advanced features and change the `Caption` of the special item from *'Short'* to *'Long'*.
5. In the Menu property of the form, set the MainMenu component you want to use when the application starts, choosing one of the two available. Note that this operation has an effect on the form at design-time, too.
6. Write the code for the Short and Long commands so that when they are selected, the menu changes.

If you follow these steps, you'll end up with an application similar to TwoMenus, which can change its menu at run-time. The example has two different MainMenu components, with useless "dummy" menu items, plus the Short and Long commands.

The application does nothing apart from changing the main menu when the Short Menu or Long Menu items are selected. Here is the code for the Short Menu item:

```
procedure TForm1.ShortMenus1Click(Sender: TObject);
begin
  {activate short menu}
  Form1.Menu := MainMenu2;
end;
```

Graphical Menu Items

Besides the run-time changes on a menu's structure I've listed so far, which are all directly available in Delphi, there are a number of operations you can perform on menus using the Windows API. In fact, there are several API functions referring to menus.

In particular, using Windows API functions for menus lets us add some graphics to them. We can customize the check mark, replace the strings with bitmaps, and even paint in the menu items.

Customizing the Menu Check Mark

As I've just mentioned, there are a number of ways to customize a menu in Windows. In this section, I'm going to show you how you can customize the check mark used by a menu item, using two bitmaps of your own. This example, NewCheck, involves using bitmaps and calling a Windows API function.

First I should explain why we need two bitmaps, not just one. If you look at a menu item, it can have either a check mark or nothing. In general, however, Windows uses two different bitmaps for the checked and unchecked menu item. I've prepared two bitmaps, of 16 x 14 pixels, using the Delphi Image Editor. You can easily run this program from the Tools menu, but you can prepare the bitmaps with any editor, including Windows Paintbrush. The bitmaps should be stored in two BMP files in the same directory as the project.

The NewCheck example has a very simple form, with just two components, a MainMenu and a label:

```
object Form1: TForm1
```

```
  Caption = 'New Check'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  object Label1: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'OFF'
    Font.Height = -96
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object MainMenu1: TMainMenu
    object Command1: TMenuItem
      Caption = '&Command'
      OnClick = Command1Click
      object Toggle1: TMenuItem
        Caption = '&Toggle'
        OnClick = Toggle1Click
      end
    end
  end
end
```

As you can see in the listing above, the menu item has a single command (Toggle), which will be used to change the text of the label from *'ON'* to *'OFF'* and change the check mark, too:

```
procedure TForm1.Toggle1Click(Sender: TObject);
begin
  Toggle1.Checked := not Toggle1.Checked;
  if Toggle1.Checked then
    Label1.Caption := 'ON'
  else
    Label1.Caption := 'OFF';
end;
```

The most important portion of the code of this example is the call to the `SetMenuItemBitmaps` Windows API function:

```
function SetMenuItemBitmaps (Menu: HMenu;
  Position, Flags: Word;
  BitmapUnchecked, BitmapChecked: HBitmap): Bool;
```

This function has a number of parameters:

- The first parameter is the pull-down menu we refer to.
- The second parameter is the position of the menu item in that pull-down menu.
- The third parameter is a flag that determines how to interpret the previous parameter (Position).
- The last two parameters indicate the bitmaps that should be used.

Notice that this function changes the check mark bitmaps only for a specific menu item. Here is the code you can use in Delphi to call the function:

```
procedure TForm1.Command1Click(Sender: TObject);
begin
  SetMenuItemBitmaps (Command1.Handle,
    Toggle1.Command, MF_BYCOMMAND,
    Bmp2.Handle, Bmp1.Handle);
end;
```

This call uses two bitmap variables that are defined in the code and the names of some components (`Command1` is the name of the pull-down, and `Toggle1` is the name of the menu item). The code above shows that it is usually very easy to pass the handle of an element to a Windows function — just use its `Handle` property.

At first I thought this function could be called when the form was created, after the two bitmaps had been loaded from the file, but it cannot. Delphi changes the default Windows behavior somewhat, forcing the application to re-associate the bitmap with the menu items each time they are displayed. The solution I've found is to execute this call each time the pull-down menu is selected — that is, on the `OnClick` event of the pull-down.

The only thing left is to load the bitmaps. You need to add two fields of the `TBitmap` type to the form class, create an instance of the two objects, and then load the bitmaps from the two `BMP` files. This is done only once, when the form is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Bmp1 := TBitmap.Create;
  Bmp2 := TBitmap.Create;
  Bmp1.LoadFromFile ('ok.bmp');
  Bmp2.LoadFromFile ('no.bmp');
end;
```

The two bitmaps should also be destroyed when the program terminates (in the handler of the `OnDestroy` event of the form):

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Bmp1.Free;
  Bmp2.Free;
end;
```

Notice that to run this program, you need to have the two `BMP` files in the same directory as the executable file. The bitmaps, in fact, are loaded at run-time and are not embedded by Delphi in the `EXE` file. As an alternative I could have used two non-visible Image components to hold the images.

> You can indeed include a bitmap in the resources of an application and in its executable file in order to be able to ship the application in a single file. This process, however, is slightly more complex, so I've decided not to use it here.

Bitmap Menu Items

Instead of placing a bitmap close to a menu item to indicate the status of its `Checked` property, as we've done in the previous section, you can actually replace the text of a menu item with a bitmap. In specific cases this can make an application easier to use.

BitMenu is a very simple program. I've put a shape in the middle of a form and added a menu to set the kind of shape (rectangle, rounded rectangle, or ellipse), and its color. Here is the textual description of the form and its menu:

```
object Form1: TForm1
  Caption = ' Bitmap Menu'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  OnResize = FormResize
  object ShapeDemo: TShape...  // default properties
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Exit1: TMenuItem...
    end
    object Shape1: TMenuItem
      Caption = '&Shape'
      object Ellipse1: TMenuItem
        Caption = 'Ellipse'
        OnClick = Ellipse1Click
      end
      object RoundRec1: TMenuItem
        Caption = 'RoundRec'
        OnClick = RoundRec1Click
      end
      object Rectang1: TMenuItem
        Caption = 'Rectang'
        OnClick = Rectang1Click
      end
    end
    object Color1: TMenuItem
      Caption = '&Color'
      object Red1: TMenuItem
        Caption = 'Red'
        OnClick = Red1Click
      end
      object Green1: TMenuItem
        Caption = 'Green'
        OnClick = Green1Click
      end
      object Blue1: TMenuItem
        Caption = 'Blue'
        OnClick = Blue1Click
      end
    end
    object Help1: TMenuItem...
    object About1: TMenuItem...
  end
end
```

I've listed the complete description of the menu items because their captions will play an important role in the code, as you'll see shortly. As a second step I've made the program work, by handling the various menu commands. Here are two examples from the two main pull-down menus:

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  ShapeDemo.Brush.Color := clRed;
end;

procedure TForm1.Ellipse1Click(Sender: TObject);
```

```
begin
  ShapeDemo.Shape := stEllipse;
end;
```

I've also written a handler for the `OnResize` event of the form, to resize the shape depending on the actual size of the form. Instead of setting its four positional properties (`Left`, `Top`, `Width`, and `Height`) I've called the `SetBounds` method. This approach leads to faster code.

Now that we've written the code of the program, it is time to turn the menu items into bitmaps. To accomplish this I've prepared a bitmap file for each of the three shapes and one for each of the three base colors. Technically these bitmaps can have any size, but to make them look nice they should generally be wide and low (the typical size of a menu item).

Having prepared the six bitmap files, each named with the caption of its menu items (plus the `.BMP` extension), I've written a handler for the `OnCreate` event of the form, and replaced the text of the items of the fake standard menu with bitmaps. This can be accomplished by calling the `ModifyMenu` API function:

```
function ModifyMenu (hMnu: HMENU;
  uPosition, uFlags, uIDNewItem: UINT;
  lpNewItem: PChar): BOOL; stdcall;
```

The first parameter is the handle of the menu we are working on (typically a pull-down menu), the second is the position of the item we want to modify, the third is some menu flags indicating the effect of other parameters, the fourth is the identifier of the menu item (stored by Delphi in the `Command` property), and the last is the new string for the menu item. How can we use this to set a bitmap, instead? We simply pass the handle of a bitmap in the last parameter (instead of the string), and use the `mf_Bitmap` menu flag among the values of the `uFlags` parameter.

Now you are ready to look at the source code, which is based on two `for` loops. I wanted to make the code as generic as possible, so that adding a new kind of shape or color will be pretty straightforward:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  // load the bitmaps for the shapes
  for I := 0 to Shape1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Shape1.Items [I].Caption + '.bmp');
    ModifyMenu (Shape1.Handle,Shape1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap,
      Shape1.Items [I].Command, Pointer (Bmp.Handle));
    Shape1.Items [I].Tag := Integer (Bmp);
  end;

  // load the bitmaps for the colors
  for I := 0 to Color1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Color1.Items [I].Caption + '.bmp');
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap, Color1.Items [I].Command,
      Pointer (Bmp.Handle));
    Color1.Items [I].Tag := Integer (Bmp);
  end;
```

```
end;
```

In the two calls to the `ModifyMenu` API function in the code above, we've used the `mf_ByPosition` flag in the third parameter, and passed the position of the item in the second parameter. However, we still need to pass the menu command as the fourth parameter, because this will be the command of the new menu item. If we don't, we'll lose the connection between the menu item and its Delphi event handler.

As you can see I save the reference to each bitmap object I create in the `Tag` property of the corresponding menu item. The only reason I have to save the bitmap objects is to be able to destroy them when the application terminates:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Shape1.Count - 1 do
    TBitmap (Shape1.Items [I].Tag).Free;
  for I := 0 to Color1.Count - 1 do
    TBitmap (Color1.Items [I].Tag).Free;
end;
```

As an alternative I could have saved the bitmap objects in an array (declared as a field of the form), and then destroyed each of the objects of the array at the end.

When you run this program, the Shape pull-down menu looks nice. The Color pull-down menu, however, is not working properly. As soon as you select it, in fact, the colors are displayed properly. But when you move over a menu item (as you can see by running the program) its colors are reversed by the menu item selection. Reversing the color you are asking for results in a very odd user interface: to select the color blue you have to click on an item that has temporarily turned to yellow!

Basically, you can use only black-and-white or gray-scaled bitmaps in a menu item. Color bitmaps create a lot of problems. If you want to obtain this effect, you should use an owner-draw menu item, as described in the next section.

Owner-Draw Menu Items

In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

Actually in Delphi the situation is slightly more complex. The components can take care of painting themselves also in this case (as is the case for the `TBitBtn` class for bitmap buttons), and eventually activate corresponding events. If you don't think about the internal details, owner-draw techniques for list boxes and combo boxes require you simply to write the handler for a couple of events.

Delphi provides no support for owner-draw menu items, though. So in this case we have to use the standard Windows approach, and handle a couple of system messages in our form. Here is the definition of these methods in the source code of the ODMenu example, an extension of the BitMenu example discussed in the last section:

```
type
```

```
  TForm1 = class(TForm)
    ...
  public
    procedure WmMeasureItem (var Msg: TWmMeasureItem);
      message wm_MeasureItem;
    procedure WmDrawItem (var Msg: TWmDrawItem);
      message wm_DrawItem;
  end;
```

The wm_MeasureItem message is sent by Windows once for each menu item when the pull-down menu is displayed to determine the size of each item. The wm_DrawItem message is sent when an item has to be repainted. This happens when Windows first displays the items, and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on.

In the ODMenu example I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've modified the CreateForm method to call the ModifyMenu API function passing the mfOwnerDraw menu flag. I also pass a code as the last parameter, to distinguish the various items. The other parameters are the same as in the previous version of the example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // turn the menu items into owner-draw items
  for I := 0 to Color1.Count - 1 do
  begin
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_OwnerDraw,
      Color1.Items [I].Command, Pointer (I));
  end;
end;
```

Now we have to write the code of the two owner-draw message handlers. The WmMeasureItem method receives as a parameter (in the corresponding TWmMeasureItem structure) a pointer to the Windows MeasureItemStruct structure (you can see the details of this structure in the Windows API help file). From this last structure we use the CtlType field, which stores the type of element we are measuring, to check whether this operation pertains to a menu. If it does, we use the ItemWidth and ItemHeight fields to set the width and height of the item. Since all items have the same size, the code is quite simple. The last thing we have to do is to provide a return value for the message, indicating we have handled it, in the Result field of the message structure. Here is the complete code:

```
procedure TForm1.WmMeasureItem (var Msg: TWmMeasureItem);
begin
  inherited;
  with Msg.MeasureItemStruct^ do
    if CtlType = odt_Menu then
    begin
      ItemWidth := 80;
      ItemHeight := 30;
      Msg.Result := 1; // we've handled it
    end;
end;
```

Drawing a menu item is slightly more complex, since to write Delphi code we have to create a `TCanvas` object, which encapsulates a Windows *device context* handle. Since we have to free this object to avoid memory leaks, I've used a `try-finally` statement to destroy it even if an error occurs:

```
Canvas1 := TCanvas.Create;
Canvas1.Handle := hDC;
try
  ...
finally
  Canvas1.Free;
end;
```

The code assigns to the `Handle` property of this `TCanvas` object the `hDC` field passed by the message in the `DrawItemStruct` structure (there is, again, a pointer to this structure passed in one of the fields of the `TWmDrawItem` parameter of the message-handler method). This `hDC` field passes to our code the handle to the device context (the painting area, the Canvas) of the pull-down menu we are going to paint.

Once we have set up the drawing mechanism, we can start with the actual code. First we have to check the state of the menu item, stored in the `ItemState` field. If this includes the `ods_Selected` flag (a condition we can test by checking the result of a bitwise `and` expression); then we have to paint the background of the menu item using the Windows system color for the highlighted items, `clHighlight`. Otherwise, we use the standard color for menus, `clMenu`.

Some of the Delphi constants for colors correspond to the Windows system color. These colors are not fixed, but reflect the current color setting made by the user.

Once we've assigned to the `Brush` of the canvas the background color, we can easily paint it by calling the `FillRect` method. This method has one single parameter, corresponding to the rectangle we have to erase. This information is available in the `rcItem` field of the `DrawItemStruct` structure. Notice, by the way, that if you don't limit your drawing area to this rectangle, you can paint over other menu items as well!

The second step is drawing the actual colored areas, which don't depend on the status of the menu item, since we want each color to show up properly even if the item is selected. By looking to the `ItemData` field, the code retrieves the code passed as the last parameter of the `ModifyMenu` function call, used to make the menu item owner-draw. Depending on the value of this field, the program sets a proper value for the `Color` of the `Brush` using a simple `case` statement. At this point we only have to paint the area, by calling the `Rectangle` function and passing, in the four parameters, a smaller area than the full surface of the menu item. In fact we need to reserve a border with the background color. In the example the border is 5 pixels wide.

Here, finally, is the complete source code of the `WmDrawItem` method:

```
procedure TForm1.WmDrawItem (var Msg: TWmDrawItem);
var
  Canvas1: TCanvas;
begin
  inherited;
  with Msg.DrawItemStruct^ do
    if CtlType = odt_Menu then
    begin
      // create a canvas for painting
      Canvas1 := TCanvas.Create;
      Canvas1.Handle := hDC;
      try
```

```
          // set the background color and draw it
          if (ods_Selected and ItemState <> 0) then
            Canvas1.Brush.Color := clHighlight
          else
            Canvas1.Brush.Color := clMenu;
          Canvas1.FillRect (rcItem);
          case ItemData of
            0: Canvas1.Brush.Color := clRed;
            1: Canvas1.Brush.Color := clLime;
            2: Canvas1.Brush.Color := clBlue;
          end;
          Canvas1.Rectangle (rcItem.Left + 5, rcItem.Top + 5,
            rcItem.Right - 10, rcItem.Bottom - 10);
        finally
          Canvas1.Free;
        end;
      end;
end;
```

I suggest you run this program along with the BitMenu program, to see the differences in the way the colored menu items are painted. Actually, looking at these two examples one might think of using the owner-draw technique also for menu items with black-and-white bitmaps.

Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and for "quick-and-dirty" applications. Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_STRING, idSysAbout, '&About...');
```

The code fragment above (extracted from the SysMenu example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```
const
  idSysAbout = 100;
```

In the SysMenu example, this code is executed in the `OnCreate` event handler, and it produces the new system menu. Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWmSysCommand` parameter) of the menu item is our `idSysAbout`. Since there isn't a corresponding event in Delphi, we have to define a new message response method to the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a MainMenu component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the `Menu` property of the form, removing the menu bar. This way we have a MainMenu component but nothing on the screen.

Now we can add some code to the SysMenu example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is pressed. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

This code uses the expression `self.Handle` to access the handle of the form. This is required because we are currently working on the `MainMenu1` component, as specified by the `with` statement.

The menu flag used in this case, `mf_Popup`, indicates that we are adding a pull-down menu. In this function call the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example we passed the identifier of the menu, instead). Since we are adding to the system menu items with sub-menus, the final structure of the system menu will have two levels.

> The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms for two different things, the local menus and the secondary menus of the menu bar. Apparently, they've done this because these two elements are implemented with the same kind of internal windows; and the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, as usual, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When we have found the menu item (and if we have found something), we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
  if Item <> nil then
    Item.Click;
```

In this code, The `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file, available in Delphi.

This application works but has one glitch. If you click the right mouse button over the TaskBar icon representing the application, you get a plain system menu (actually even different than the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object.

Building a Complete Menu

Now that we know how to write a menu, disable and check menu items, and so on, we are ready to build the menu for a full-fledged application. Do you remember the RichNote example of the last chapter? It was a simple editor based on a RichEdit component. You could use it to write and change the font of the selected text, but that was all. Now we want to add a menu and implement a number of features, including a complete scheme for opening and saving the text files. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

First of all, we need to build the menu, following the standard. The main menu starts with two standard pull-down menus, File and Edit, with the typical menu items. Then there are two specific pull-down menus, Font and Paragraph, with menu items to set the text font and alignment. The last two pull-down menus, Options and Help, are *almost* standard: Their names are standard, but their menu items are not. The Options menu has commands to change the background color and to count the characters, and the Help menu has only the About menu item.

In this example, we want to implement most but not all of the commands of the menu, for example skipping the Clipboard commands. The following table shows the complete structure of the menu:

&File	&Edit	F&ont	&Paragraph	&Options	&Help
&New	Cu&t	&Times New Roman	&Left Aligned	&Backgroud Color	&About RichNote...
&Open...	&Copy	&Courier New	&Right Aligned	&Read Only	
&Save	&Paste	&Arial	&Centered	&Count chars...	
Save&As...		&Bold			
&Print...		&Italic			
E&xit		&Small			
		&Medium			
		&Large			
		More &Fonts...			

Having added a complete menu, we can now get rid of the simple panel and the font button of the RichNote example. The only visual component left in the form of the RichNot2 version will be the RichEdit component, which is aligned with the client area. Then there is the MainMenu component, and four components for standard dialog boxes (OpenDialog, SaveDialog, FontDialog, and ColorDialog).

The File Menu

As I mentioned when we began working through the RichNot2 example, the most complex part of this program is implementing the commands of the File pull-down menu—New, Open, Save, and Save As. In each case, we need to track whether the current file has changed, saving the file only if it has. We should prompt the user to save the file each time the program creates a new file, loads an existing one, or terminates.

To accomplish this, I've added two fields and three methods to the class describing the form of the application:

```
  private
    FileName: string;
    Modified: Boolean;
  public
    function SaveChanges: Boolean;
    function Save: Boolean;
    function SaveAs: Boolean;
```

The `FileName` string and the `Modified` flag are set when the form is created and changed when a new file is loaded or the user renames a file with the Save As command. These two flags are initialized when the form is first created:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
begin
  FileName := '';
  Modified := False;
end;
```

The value of the flag changes as soon as you type new characters in the RichEdit control (in its `OnChange` event handler):

```
procedure TFormRichNote.RichEdit1Change(Sender: TObject);
begin
  Modified := True;
end;
```

When a new file is created, the program checks whether the text has been modified. If so, it calls the SaveChanges function, which asks the user whether to save the changes, discard them, or skip the current operation:

```
procedure TFormRichNote.New1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
  begin
    RichEdit1.Text := '';
    Modified := False;
    FileName := '';
    Caption := 'RichNote - [Untitled]';
  end;
end;
```

If the creation of a new file is confirmed, some simple operations take place, including using *'Untitled'* instead of the file name in the form's caption.

Short-Circuit Evaluation

The expression **if not** Modified **or** SaveChanges **then** requires some explanation. By default, Pascal performs what is called "short-circuit evaluation" of complex conditional expressions. The idea is simple: if the expression not Modified is true, we are sure that the whole expression is going to be true, and we don't need to evaluate the second expression. In this particular case, the second expression is a function call, and the function is called only if Modified is True. This behavior of or and and expressions can be changed by setting a Delphi compiler option called Complete Boolean Eval. You can find it on the Compiler page of the Project Options dialog box.

The message box displayed by the SaveChanges function has three options. If the user selects the Cancel button, the function returns False. If the user selects No, nothing happens (the file is not saved) and the function returns True, to indicate that although we haven't actually saved the file, the requested operation (such as creating a new file) can be accomplished. If the user selects Yes, the file is saved and the function returns True.

In the code of this function, notice in particular the call to the MessageDlg function used as the value of a case statement:

```
function TFormRichNote.SaveChanges: Boolean;
begin
  case MessageDlg (
    'The document ' + filename + ' has changed.' +
    #13#13 + 'Do you want to save the changes?',
    mtConfirmation, mbYesNoCancel, 0) of
  idYes:
    // call Save and return its result
    Result := Save;
  idNo:
    // don't save and continue
    Result := True;
  else // idCancel:
    // don't save and abort operation
    Result := False;
  end;
```

```
end;
```

In the MessageDlg call above, I've added explicit newline characters (`#13`) to improve the readability of the output. As an alternative to using a numeric character constant, you can call `Chr(13)`.

To actually save the file, another function is invoked: `Save`. This method saves the file if it already has a proper file name or asks the user to enter a name, calling the `SaveAs` functions. These are two more internal functions, not directly connected with menu items:

```
function TFormRichNote.Save: Boolean;
begin
  if Filename = '' then
    Result := SaveAs // ask for a file name
  else
  begin
    RichEdit1.Lines.SaveToFile (FileName);
    Modified := False;
    Result := True;
  end;
end;

function TFormRichNote.SaveAs: Boolean;
begin
  SaveDialog1.FileName := Filename;
  if SaveDialog1.Execute then
  begin
    Filename := SaveDialog1.FileName;
    Save;
    Caption := 'RichNote - ' + Filename;
    Result := True;
  end
  else
    Result := False;
end;
```

I use two functions to perform the Save and SaveAs operations (and do not call the corresponding menu handler directly) because I need a way to report a request to cancel the operation from the user. To avoid code duplication, the handlers of the Save and SaveAs menu items call the two functions too, although they ignore the return value:

```
procedure TFormRichNote.Save1Click(Sender: TObject);
begin
  if Modified then
    Save;
end;

procedure TFormRichNote.Saveas1Click(Sender: TObject);
begin
  SaveAs;
end;
```

Opening a file is much simpler. Before loading a new file, the program checks whether the current file has changed, asking the user to save it with the `SaveChanges` function, as before. The `Open1Click` method is based on the OpenDialog component, another default dialog box provided by Windows and supported by Delphi:

```
procedure TFormRichNote.Open1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
    if OpenDialog1.Execute then
    begin
      Filename := OpenDialog1.FileName;
      RichEdit1.Lines.LoadFromFile (FileName);
      Modified := False;
      Caption := 'RichNote - ' + FileName;
    end;
end;
```

The only other detail related to file operations is that both the OpenDialog and SaveDialog components of the NotesForm have a particular value for their `Filter` and `DefaultExt` properties, as you can see in the following fragment from the textual description of the form:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'rtf'
  FileEditStyle = fsEdit
  Filter = 'Rich Text File (*.rtf)|*.rtf|Any file (*.*)|*.*'
  Options = [ofHideReadOnly, ofPathMustExist,ofFileMustExist]
end
```

The string used for the `Filter` property (which should be written on a single line) contains four pairs of substrings, separated by the | symbol. Each pair has a description of the type of file that will appear in the File Open or File Save dialog box, and the filter to be applied to the files in the directory, such as `*.RTF`. To set the filters in Delphi, you can simply invoke the editor of this property, which displays a list with two columns.

The file-related methods above are also called from the `FormCloseQuery` method (the handler of the `OnCloseQuery` event), which is called each time the user tries to close the form, terminating the program. We can make this happen in various ways — by double-clicking on the system menu icon, selecting the system menu's Close command, pressing the Alt+F4 keys, or calling the `Close` method in the code, as in the File | Exit menu command.

In `FormCloseQuery`, you can decide whether or not to actually close the application by setting the `CanClose` parameter, which is passed by reference. Again, if the current file has been modified, we call the `SaveChanges` function and use its return value. Again we can use the short-circuit evaluation technique:

```
procedure TFormRichNote.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Modified or SaveChanges;
end;
```

The last menu item of the File menu is the Print command. Since the RichEdit component includes print capabilities and they are very simple to use, I've decided to implement it anyway. Here is the code, which actually produces a very nice printout:

```
procedure TFormRichNote.Print1Click(Sender: TObject);
begin
  RichEdit1.Print (FileName);
end;
```

The Paragraph Menu

Compared to the File menu, the other pull-down menus of this example are simpler. The code of the Paragraph menu is based on some properties of its items. Here is their textual description (from the DFM file):

```
object Paragraph1: TMenuItem
  Caption = '&Paragraph'
  object LeftAligned1: TMenuItem
    Caption = '&Left Aligned'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object RightAligned1: TMenuItem
    Caption = '&Right Aligned'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object Centered1: TMenuItem
    Caption = '&Centered'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
end
```

As you can see, the program uses radio menu items, by giving to the three items the same value for the `GroupIndex` property and setting the `RadioItem` property to `True`. The menu items also share the same `RightAligned1Click` method for their `OnClick` event. Here is the code of the method, which is based on the correspondence between the position of the menu items in the pull-down (indicated by their `MenuIndex` property) and the order of the values of the `TAlignment` enumeration. It is a trick, but it works. Here is the code:

```
procedure TFormRichNote.RightAligned1Click(Sender: TObject);
begin
  RichEdit1.Paragraph.Alignment :=
    TAlignment ((Sender as TMenuItem).MenuIndex);
  (Sender as TMenuItem).Checked := True;
end;
```

First, this procedure sets the alignment of the current paragraph (the paragraph including the selected text or the editor cursor), then it checks the current menu item — the menu item that has activated the method (the `Sender` object). As you can see, this code relies on some controlled typecasts, based on the `as` keyword: this is what you have to do any time you want to write generic code (that is, to attach the same methods to events of different components).

Notice that setting the check mark for the current menu item is correct only until you change the selection or the current line in the text. For this reason, we can handle the `OnSelectionChange` event of the RichEdit component, and update the check mark of the Paragraph menu each time:

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
begin
  Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
    Checked := True;
```

```
end;
```

The Font Menu

The Font pull-down menu is built on the same concept, but it has two groups of radio items, plus items with a standard check mark. Each group, then, uses a different approach to handle the menu item selection with a single event response method. You can see the details of the properties of the menu items directly in the `RichForm.DFM` source code file (this listing was too long to reproduce here).

The code of the first group of menu items (used to select the font) is based on a simple trick: the name of the font to select corresponds to the `Caption` of the menu item, without the initial `&` character:

```
procedure TFormRichNote.TimesRoman1Click(Sender: TObject);
var
  FontName: string;
begin
  // get the font name and remove the &
  FontName := (Sender as TMenuItem).Caption;
  Delete (FontName, 1, 1);
  // change selected text font
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Name := FontName;
  (Sender as TMenuItem).Checked := True;
end;
```

This code acts on the current selection (using the `SelAttributes` property of the RichEdit1 component), as the RichNote example in Chapter 8 did. Notice that you can easily extend this code by adding new menu items consisting of the name of the font preceded by the `&` character. Then you'll necessarily have to set the same value for the `GroupIndex` property of the previous items (in this case `1`). So if you simply set the `RadioItem` property to `True` and connect the `OnClick` event to the `TimesRoman1Click` method, they'll behave just like the existing font selection menu items.

The second group of items controls the selection of the bold and italic styles. This is accomplished by two similar but separate methods. Here is one of them:

```
procedure TFormRichNote.Bold1Click(Sender: TObject);
begin
  Bold1.Checked := not Bold1.Checked;
  if RichEdit1.SelLength > 0 then
    with RichEdit1.SelAttributes do
      if Bold1.Checked then
        Style := Style + [fsBold]
      else
        Style := Style - [fsBold];
end;
```

The last part of the menu has another group of radio menu items, used to set the size of the font. These menu items refer to a font size in the caption and have the same value stored in their `Tag` property. This makes the code very easy to write. Again, there is a single method for the three menu items, but you can add new items to this group with very little effort:

```
procedure TFormRichNote.Large1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Size :=(Sender as TMenuItem).Tag;
  (Sender as TMenuItem).Checked := True;
end;
```

The last item of this menu simply activates the Font dialog box. Notice that the font returned by this dialog box cannot be assigned directly to the `SelAttributes` property; we need to call the `Assign` method, instead:

```
procedure TFormRichNote.More1Click(Sender: TObject);
begin
  FontDialog1.Font := RichEdit1.Font;
  if FontDialog1.Execute then
    RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  // update the check marks
  RichEdit1SelectionChange (self);
end;
```

All the commands of this menu affect the status of the current selection, setting the check boxes and radio items properly. The method above, instead, calls the `RichEdit1SelectionChange` method. This is the handler of the `OnSelectionChange` event of the RichEdit component.

This method scans some of the groups to determine which element has to be checked. As in other examples before, this code has been written so that you can easily extend it for new menu items (the listing also includes the code used to reset the paragraph alignment, discussed earlier):

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
var
  FontName: string;
  I: Integer;
begin
  // check the font name radio menu item
  FontName := '&' + RichEdit1.SelAttributes.Name;
  for I := 0 to 2 do
    with Font1.Items [I] do
      if FontName = Caption then
        Checked := True;

  // check the bold and italic items
  Italic1.Checked :=
    fsItalic in RichEdit1.SelAttributes.Style;
  Bold1.Checked :=
    fsBold in RichEdit1.SelAttributes.Style;

  // check the font size
  for I := Small1.MenuIndex to Large1.MenuIndex do
    with Font1.Items [I] do
      if Tag = RichEdit1.SelAttributes.Size then
        Checked := True;

  // check the paragraph style
  Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
    Checked := True;
end;
```

This method doesn't work perfectly. When you set a custom font, the selection doesn't change properly (because the current item remains selected). To fix it, we should remove any radio check mark when the value is not one of the possible selections, or add new menu items for this special case. The example is complex enough, so I think we can live with this minor inconvenience.

The Options Menu

The last pull-down menu of the RichNot2 example is the Options menu. This menu has three unrelated commands used to customize the user interface, and to determine and display the length of the text. The first command displays a color selection dialog box, used to change the color of the background of the RichEdit component:

```
procedure TFormRichNote.BackColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := RichEdit1.Color;
  if ColorDialog1.Execute then
    RichEdit1.Color := ColorDialog1.Color;
end;
```

The second command can be used to mark the text as read-only. In theory this property should be set depending on the status of the file on the disk. In the example, instead, the user can toggle the read-only attribute manually:

```
procedure TFormRichNote.ReadOnly1Click(Sender: TObject);
begin
  RichEdit1.ReadOnly := not RichEdit1.ReadOnly;
  ReadOnly1.Checked := not ReadOnly1.Checked;
end;
```

The last menu item activates a method to count the number of characters in the text and display the total in a message box. The core of the method is the call to the `GetTextLen` function of the RichEdit control. The number is extracted and formatted into an output string:

```
procedure TFormRichNote.Countchars1Click(Sender: TObject);
begin
  MessageDlg (Format (
    'The text has %d characters', [RichEdit1.GetTextLen]),
    mtInformation, [mbOK], 0);
end;
```

The handler of the `OnClick` event of this last item of the Options menu terminates the example. As mentioned at the beginning, this was a rather long and complex example, but its purpose was to show you the implementation of the menu commands of a real-world application. In particular, I explained in detail the File pull-down menu because this is something you'll probably need to handle in any file-related application.

Now we are ready to delve into another topic involving menus, the use of local menus activated by the right mouse button click.

Pop-Up Menus

In Windows, it is common to see applications that have special local menus you activate by clicking the right mouse button. The menu that is displayed — a pop-up menu, in common Windows terminology — usually depends on the position of the mouse click. These menus tend to be easy to use since they group only the few commands related to the element that is currently selected. They are also usually faster to use than full-blown menus because you don't need to move the mouse up to the menu bar and then down again to go on working.

In Delphi, there are basically two ways to display pop-up menus, using the corresponding component. You can let Delphi handle them automatically or you can choose a manual technique. I'll explore both approaches, starting with the first, which is the simplest one.

To add a pop-up menu to a form, you need to perform a few simple operations. Create a PopupMenu component, add some menu items to it, and select the component as the value of the form's `PopupMenu` property. That's all. Of course, you should also add some handlers for the `OnClick` events of the local menu's various menu items, as you do with an ordinary menu.

An Automatic Local Menu

To show you how to create a local menu, I've built an example that is an extension of the Dragging example. The new example is named Local1. I've added a first PopupMenu component to its form and connected it using the `PopupMenu` property of the form itself. Once this is done, running the program and clicking the right mouse button on the form displays the local menu. Then, I've added a second pop-up menu component, with two levels, to the form, and I've attached it to the StaticText component on the right, `LabelTarget`. To connect a local menu to a specific component, you simply need to set its `PopupMenu` property. The four methods related to the first group of commands of the Colors pull-down menu just select a color:

```
procedure TDraggingForm.Aqua1Click(Sender: TObject);
begin
  LabelTarget.Color := clAqua;
end;
```

The Transparent command selects the color of the parent form as the current color, setting the value of the `ParentColor` property to `True`.

The components of a form usually borrow some properties from the form. This is indicated by specific properties, such as `ParentColor` or `ParentFont`. When these properties are set to `True`, the current value of the component's property is ignored, and the value of the form is used instead. Usually, this is not a problem, because as soon as you set a property of the component (for example, the font), the corresponding property indicating the use of the parent attribute (`ParentFont`) is automatically set to `False`.

The last command of the pull-down menu, User Defined, presents the standard Color Selection dialog box to the user. The three commands of the pop-up menu's second pull-down change the alignment of the text of the big label and add a check mark near the current selection, deselecting the other two menu items. Here is one of the three methods:

```
procedure TDraggingForm.Center1Click(Sender: TObject);
begin
  LabelTarget.Alignment := taCenter;
  Left1.Checked := False;
  Center1.Checked := True;
  Right1.Checked := False;
end;
```

A pop-up menu, in fact, can use all the features of a main menu and can have checked, disabled, or hidden items, and more. I could have also used radio menu items for this pop-up menu, but this doesn't seem to be a very common approach.

Modifying a Pop-Up Menu When It Is Activated

Why not use the same technique to display a check mark near the selected color? It is possible, but it's not a very good solution. In fact, there are six menu items to consider, and the color can also change when a user drags it from one of the labels on the left of the form. For this reason, and to show you another technique, I've followed a different approach.

Each time a pop-up menu is displayed, the `OnPopup` event is sent to your application. In the code of the corresponding method, you can place the check mark on the current selection of the color, independently from the action used to set it:

```
procedure TDraggingForm.PopupMenu2Popup(Sender: TObject);
var
  I: Integer;
begin
  // unchecks all menu items (not required for radio menu items)
  with Colors1 do
    for I := 0 to Count - 1 do
      Items[I].Checked := False;

  // checks the proper item
  case LabelTarget.Color of
    clRed: Red1.Checked := True;
    clAqua: Aqua1.Checked := True;
    clGreen: Green1.Checked := True;
    clYellow: Yellow1.Checked := True;
  else
    if LabelTarget.ParentColor then
      Transparent1.Checked := True
    else
      UserDefined1.Checked := True;
  end;
end;
```

This method's code requires some explanation. At the beginning, the menu items are all unchecked by using a `for` loop on the `Items` array of the Colors1 menu. The advantage of this loop is that it operates on all the menu items, regardless of their number. Then the program uses a `case` statement to check the proper item.

Handling Pop-Up Menus Manually

In the Local1 example, we saw how to use automatic pop-up menus. As an alternative, you can set the `AutoPopup` property to `False` or not connect the pop-up menu to any component, and use the pop-up menu's `Popup` method to display it on the screen. This procedure requires two parameters: the *x* and *y* values of the position where the menu is going to be displayed. The problem is that you need to supply the *screen* coordinates of the point, not the client coordinates, which are the usual coordinates relative to the form's client area.

As an example, I've taken an existing application with a menu — the third version of the MenuOne example, described in this chapter — and added a peculiar pop-up menu. The idea is that there are two different pop-up menus, one to change the colors and the other to change the alignment of the text. Each time the user right-clicks on the caption, one of the two pop-up menus is displayed. In real applications, you'll probably have to decide which menu to display depending on the status of some variable. Here, I've followed a simple (and arbitrary) rule: Each time the right mouse button is clicked, the pop-up menu changes. My aim is to show you how to do this in the simplest possible way.

The two pop-up menus are very simple, and correspond to actions already available in the main menu (and connected with the same event handlers). Actually, I've built these pop-up menus by copying the main menu to a menu template and then pasting from it. The only change I've made is to remove the shortcut keys.

When the user clicks the right mouse button over the label, which takes up the whole surface of the form, a method displays one of the two menus. Instead of using the `OnClick` event, I've trapped the `OnMouseDown` event of the label. This second event passes as parameters the coordinates of the mouse click.

These coordinates are relative to the label, so you have to convert them to screen coordinates by calling the `ClientToScreen` method of the label:

```
procedure TFormColorText.Label1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ClientPoint, ScreenPoint: TPoint;
begin
  if Button = mbRight then
  begin
    ClientPoint.X := X;
    ClientPoint.Y := Y;
    ScreenPoint := Label1.ClientToScreen (ClientPoint);
    Inc (ClickCount);
    if Odd (ClickCount) then
      PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
    else
      PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  end;
end;
```

In this procedure, you first have to check whether the right mouse button was clicked. The second step is to translate the coordinate of the position of the mouse click from client coordinates to screen coordinates. Screen coordinates are required by the PopupMenu component's `Popup` method.

The last thing we have to do is provide the proper check marks for the menu items of the second pop-up menu. A solution is to copy the current check marks of the main menu to the pop-up before displaying it:

```
if Odd (ClickCount) then
  PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
else
begin
  {set the check marks as in the main menu}
  Left2.Checked := Left1.Checked;
  Center2.Checked := Center1.Checked;
  Right2.Checked := Right1.Checked;
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
end;
```

An alternative solution—the one I've actually implemented in the Local2 example — is to set this check mark every time you set the check marks of the main menu, as in the following code:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Left2.Checked := True;
end;
```

In a more general application you might want to write a generic routine to apply the check marks more consistently, but in this example one of these two simple techniques will do.

What's Next

In this chapter, we have seen how to create main menus and pop-up menus in Delphi. We've discussed the standard guidelines for the names of the pull-down menus and of the menu items, shortcut keys, check marks, graphical menus, local menus, and many other topics. You can explore in other directions, as well. For example, you can create a menu dynamically (at run-time) or copy portions of a menu to another menu, as in the SysMenu example.

The next step, however, is to explore a less common but very nice feature of Delphi programming, multimedia.

CHAPTER 6: MULTIMEDIA FUN

- Windows default sounds
- From a beep to music
- The Media Player component
- Playing sounds and running videos
- Applications for audio CD drives

Among the many devices, the multimedia subsystem focuses sound cards or CD-ROM drives. Of course, besides being physically connected to your computer, these devices must be properly installed in Windows for your Delphi applications to access them.

Windows provides a specific API, known as the *Multimedia API*, to handle external devices such as video, MIDI, and CD drives. Delphi includes a corresponding Help file along with an easy-to-use component, the Media Player, to manipulate most multimedia devices. Before discussing this component, which is the main topic of the chapter, we'll look at some simpler ways to produce sound in Windows, beyond the simple beeps we have used previously.

Windows Default Sounds

In the book's earlier examples, every time we wanted to notify the user of an error or a specific event, we called a Delphi system procedure (`Beep`) or a Windows API function (`MessageBeep`). The `Beep` procedure is defined in the Delphi run-time library as follows:

```
procedure Beep;
begin
  MessageBeep(0);
end;
```

It simply passes the value 0 to the `MessageBeep` API function. Besides the values 0 and `-1`, both used to produce a beep with the internal speaker of the computer, the `MessageBeep` function can also accept other values, and play the corresponding sounds with your sound board. Here are the acceptable constants and the corresponding Windows sounds they produce (these are the sound names available in Control Panel):

`mb_IconAsterisk`	SystemAsterisk sound
`mb_IconExclamation`	SystemExclamation sound
`mb_IconHand`	SystemHand sound
`mb_IconQuestion`	SystemQuestion sound
`mb_Ok`	SystemDefault sound

You can change the association between system events and sound files using the Control Panel, which lists the sounds under the names shown in the right column above. These associations are stored in the Windows System Registry.

Notice that these constants are also the possible values of the MessageBox API function, encapsulated in the MessageBox method of the TApplication class. It is common to produce the corresponding sound when the message box is displayed. This feature is not directly available in the Delphi MessageDlg function (which displays a message box with a corresponding icon), but we can extend it easily by building a SoundMessageDlg function, as demonstrated by the following example.

Every Box Has a Beep

To show you the capabilities of the MessageBeep API function, I've prepared a simple example, Beeps. The form of this example has a RadioGroup with some radio buttons from which the user can choose one of the five valid constants of the MessageBeep function. Here is the definition of the RadioGroup, from the textual description of the form:

```
object RadioGroup1: TRadioGroup
  Caption = 'Parameters'
  ItemIndex = 0
  Items.Strings = (
    'mb_IconAsterisk'
    'mb_IconExclamation'
    'mb_IconHand'
    'mb_IconQuestion'
    'mb_Ok')
end
```

The program plays the sound corresponding to the current selection when the user clicks on the Beep Sound button (one of the push buttons of the form). This button's OnClick event-handler first determines which radio button was selected, using a case statement, and then plays the corresponding sound:

```
procedure TForm1.BeepButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
    4: BeepConstant := mb_Ok;
  else
    BeepConstant := 0;
  end;
  MessageBeep (BeepConstant);
end;
```

The else clause of the case statement is provided mainly to prevent an annoying (but not dangerous) compiler warning. To compare the selected sound with the default beep sound, click on the second button of the column (labeled *Beep –1*), which has the following code:

```
procedure TForm1.BeepOneButtonClick(Sender: TObject);
begin
  MessageBeep (Cardinal (-1));
end;
```

You can also pass the corresponding $FFFFFFFF hexadecimal value to the MessageBeep function. There is actually no difference between the two approaches. To test whether a sound driver is installed in your system (with or without a sound card, since it is possible to have a sound driver for the PC speaker), click on the first button (labeled *Test*), which uses a multimedia function, `WaveOutGetNumDevs`, to perform the test:

```
procedure TForm1.TestButtonClick(Sender: TObject);
begin
  if WaveOutGetNumDevs > 0 then
    SoundMessageDlg ('Sound is supported',
      mtInformation, [mbOk], 0)
  else
    SoundMessageDlg ('Sound is NOT supported',
      mtError, [mbOk], 0);
end;
```

To compile this function, you need to add the MmSystem unit to the `uses` clause. If your computer has no sound driver installed, you will hear only standard beeps, regardless of which sound is selected. The last two buttons have a similar aim: they both display a message box and play the corresponding sound.

The `OnClick` event handler of the Message Box button uses the traditional Windows approach. It calls the `MessageBeep` function and then the `MessageBox` method of the `Application` object soon afterward. The effect is that the sound is 1played when the message box is displayed. In fact (depending on the sound driver), playing a sound doesn't usually stop other Windows operations. Here is the code related to this fourth button:

```
procedure TForm1.BoxButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
  else {including 4:}
    BeepConstant := mb_Ok;
  end;
  MessageBeep (BeepConstant);
  Application.MessageBox (
    PChar (RadioGroup1.Items [RadioGroup1.ItemIndex]),
    'Sound', BeepConstant);
end;
```

If you click on the last button, the program calls the `SoundMessageDlg` function, which is not an internal Delphi function. It's one I've added to the program, but you can use it in your applications. The only suggestion I have is to choose a shorter name if you want to use it frequently. `SoundMessageDlg` plays a sound, specified by its `AType` parameter, and then displays the Delphi standard message box:

```
function SoundMessageDlg (const Msg: string;
  AType: TMsgDlgType; AButtons: TMsgDlgButtons;
  HelpCtx: Longint): Integer;
var
  BeepConstant: Cardinal;
begin
  case AType of
    mtWarning: BeepConstant := mb_IconExclamation;
    mtError: BeepConstant := mb_IconHand;
    mtInformation: BeepConstant := mb_IconAsterisk;
    mtConfirmation: BeepConstant := mb_IconQuestion;
  else
    BeepConstant := mb_Ok;
  end;
  MessageBeep(BeepConstant);
  Result := MessageDlg (Msg, AType,
    AButtons, HelpCtx);
end;

procedure TForm1.MessDlgButtonClick(Sender: TObject);
var
  DlgType: TMsgDlgType;
begin
  case RadioGroup1.ItemIndex of
    0: DlgType := mtInformation;
    1: DlgType := mtWarning;
    2: DlgType := mtError;
    3: DlgType := mtConfirmation;
  else {including 4:}
    DlgType := mtCustom;
  end;
  SoundMessageDlg (
    RadioGroup1.Items [RadioGroup1.ItemIndex],
    DlgType, [mbOK], 0);
end;
```

`SoundMessageDlg` is a simple function, but your programs can really benefit from its use.

From Beeps to Music

When you use the `MessageBeep` function, your choice of sounds is limited to the default system sounds. Another Windows API function, `PlaySound`, can be used to play a system sound, as well as any other waveform file (`WAV`). Again, I've built a simple example to show you this approach. The example is named ExtBeep (for Extended Beep) and has the simple form.

The form's list box shows the names of some system sounds and some `WAV` files, available in the current directory (that is, the directory containing `ExtBeep.exe` itself). When the user clicks on the Play button, the `PlaySound` function (defined in the MmSystem unit) is called:

```
procedure TForm1.PlayButtonClick(Sender: TObject);
begin
  PlaySound (PChar (Listbox1.Items [ListBox1.ItemIndex]),
    0, snd_Async);
end;
```

The first parameter is the name of the sound—either a system sound, a `WAV` file, or a specific sound resource (see the Win32 API Help file for details). The second parameter specifies where to look for a resource sound, and the third contains a series of flags, in this case indicating that the function should return immediately and let the sound play asynchronously. (An alternative value for this parameter is `snd_Sync`. If you use this value, the function won't return until the sound has finished playing.) With asynchronous play, you can interrupt a long sound by calling the `PlaySound` function again, using `nil` for the first parameter:

```
procedure TForm1.StopButtonClick(Sender: TObject);
begin
  PlaySound (nil, 0, 0);
```

```
end;
```

This is the code executed by the ExtBeep example when the user clicks on the Stop button. This button is particularly useful for stopping the repeated execution of the sound started by the Loop button, which calls `PlaySound` passing as its last parameter `(snd_Async or snd_Loop)`.

The only other method of the example, `FormCreate`, selects the first item of the list box at startup, by setting its `ItemIndex` property to 0. This avoids run-time errors if the user clicks on the button before selecting an item from the list box. You can test this example by running it, and by adding the names of the other `WAV` files or system sounds (as listed in the registration database). I suggest you also test other values for the third parameter of the function (see the API help files for details).

The Media Player Component

Now let's move back to Delphi and use the Media Player component. The Delphi `TMediaPlayer` class encapsulates most of the capabilities of the Windows Media Control Interface (MCI), a high-level interface for controlling internal and external media devices.

Perhaps the most important property of the `TMediaPlayer` component is `DeviceType`. Its value can be `dtAutoSelect`, indicating that the type of the device depends on the file extension of the current file (the `FileName` property). As an alternative, you can select a specific device type, such as `dtAVIVideo`, `dtCDAudio`, `dtWaveAudio`, and many others.

Once the device type (and eventually the file) have been selected, you can open the corresponding device (or set `AutoOpen` to `True`), and the buttons of the Media Player component will be enabled. The component has a number of buttons, not all of which are appropriate for each media type. There are actually three properties referring to the buttons: `VisibleButtons`, `EnabledButtons`, and `ColoredButtons`. The first determines which of the buttons are present in the control, the second determines which buttons are enabled, and the third determines which buttons have colored marks. By using the first two of these properties, you can permanently or temporarily hide or disable some of the buttons.

The component has several events. The `OnClick` event is unusual because it contains one parameter indicating which button was pressed and a second parameter you can use to disable the button's default action. The `OnNotify` event later tells the component whether the action generated by the button was successful. Another event, `OnPostClick`, is sent either when the action starts or when it ends, depending on the value of the `Wait` property. This property determines whether the operation on the device should be synchronous.

Playing Sound Files

Our first example using the Media Player is very simple. The form of the MmSound example has some labels describing the current status, a button to select a new file, an OpenDialog component, and a Media Player component with the following settings:

```
object MediaPlayer1: TMediaPlayer
  VisibleButtons = [btPlay, btPause,
    btStop, btNext, btPrev]
  OnClick = MediaPlayer1Click
  OnNotify = MediaPlayer1Notify
```

```
end
```

When a user opens a new file, a wave table, or a MIDI file, the program enables the Media Player, and you can play the sound and use the other buttons, too:

```
procedure TForm1.NewButtonClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    FileLabel.Caption := OpenDialog1.Filename;
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.Notify := True;
  end;
end;
```

Since I set the `Notify` property to `True`, the Media Player invokes the corresponding event handler, which outputs the information to a label:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  case MediaPlayer1.NotifyValue of
    nvSuccessful : NotifLabel.Caption := 'Success';
    nvSuperseded : NotifLabel.Caption := 'Superseded';
    nvAborted    : NotifLabel.Caption := 'Aborted';
    nvFailure    : NotifLabel.Caption := 'Failure';
  end;
  MediaPlayer1.Notify := True;
end;
```

Notice that you need to set the `Notify` property to `True` every time the `OnNotify` event handler is called in order to receive further notifications. Another label is updated to display the requested command.

```
procedure TForm1.MediaPlayer1Click(Sender: TObject;
  Button: TMPBtnType; var DoDefault: Boolean);
begin
  case Button of
    btPlay: ActionLabel.Caption := 'Playing';
    btPause: ActionLabel.Caption := 'Paused';
    btStop: ActionLabel.Caption := 'Stopped';
    btNext: ActionLabel.Caption := 'Next';
    btPrev: ActionLabel.Caption := 'Previous';
  end;
end;
```

Running Videos

So far, we have worked with sound only. Now it is time to move to another kind of media device: video. You indeed have a video device on your system, but to play video files (such as `AVI` files), you need a specific driver (directly available in Windows). If your computer can display videos, writing a Delphi application to do so is almost trivial: place a Media Player component in a form, select an `AVI` file in the `FileName` property, set the `AutoOpen` property to `True`, and run the program. As soon as you click on the Play button, the system opens a second window and shows the video in it.

Instead of playing the file in its own window, we can add a panel (or any other windowed component) to the form and use the name of this panel as the value of the Media Player's `Display` property. As an alternative, we can set the `Display` and the `DisplayRect` properties to indicate which portions of the output window the video should cover.

Although it is possible to create a similar program writing no code at all, to do so I would have to know which `AVI` files reside on your computer, and specify the full path of one of them in the `FileName` property of the Media Player component. As an alternative, I've written a simple routine to open and start playing a file automatically. You only have to click on the panel (as the caption suggests).

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.FileName := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.Perform (wm_LButtonDown, 0, $00090009);
    MediaPlayer1.Perform (wm_LButtonUp, 0, $00090009);
  end;
end;
```

After opening the Media Player, I could have called its `Play` method immediately to start it. But that would not have enabled and disabled the buttons properly. So I decided to simulate a click in position 9 on the *x*-axis and 9 on the *y*-axis of the Media Player window (instead of building the 32-bit value including both coordinates with a function, you can use the hexadecimal value directly, as in the code above). To avoid errors, I disabled all the buttons at design-time, until the simulated click takes place. I also automatically close the player when the application is closed (in the `OnClose` event handler).

A Video in a Form

The Media Player component has some limits regarding the window it can use to produce the output. You can use many components, but not all of them. A strange thing you can try is to use the Media Player component itself as the video's output window. This works, but there are two problems. First, the Media Player component cannot be aligned, and it cannot be sized at will. If you try to use big buttons, their size will be reduced automatically at run-time. The second problem is that if you click on the Pause button, you'll see the button in front of the video, while the other buttons are still covered. (I suggest you try this approach, anyway, just for fun.)

One thing you cannot do easily is display the video in a form. In fact, although you cannot set the form as the value of the Media Player's `Display` property at design-time, you can set it at run-time. To try this, simply place a hidden Media Player component (set the `Visible` property to `False`) and an `OpenDialog` component in a form. Set a proper title and hint for the form itself, and enable the `ShowHints` property. Then write the following code to load, start, and stop the video when the user clicks on the form:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if MediaPlayer1.FileName = '' then
    if OpenDialog1.Execute then
    begin
      MediaPlayer1.FileName := OpenDialog1.FileName;
      MediaPlayer1.Open;
      Playing := False;
    end
    else
      exit; // stop if no file is selected
  if Playing then
  begin
    MediaPlayer1.Stop;
    Playing := False;
    Caption := 'MM Video (Stopped)';
    Hint := 'Click to play video';
  end
  else
  begin
    MediaPlayer1.Display := self;
    MediaPlayer1.DisplayRect := ClientRect;
    MediaPlayer1.Play;
    Playing := True;
    Caption := 'MMV (Playing)';
    Hint := 'Click to stop video';
  end;
end;
```

In this code, `Playing` is a private `Boolean` field of the form. Notice that the program shows the video using the full client area of the form. If the form is resized, you can simply enlarge the output rectangle accordingly:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  MediaPlayer1.DisplayRect := ClientRect;
end;
```

The best way to view a video is to use its original size, but with this program you can actually stretch it, and even change its proportions. Of course, the Media Player can also stop when it reaches the end of a file or when an error occurs. In both cases, we receive a notification event:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  Playing := False;
  Caption := 'MMV (Stopped)';
  Hint := 'Click to play video';
end;
```

Working with a CD Drive

In addition to audio and video files, the MCI interface is generally used to operate external devices. There are many examples, but the most common MCI device connected to a PC is probably a CD-ROM drive. Most CD-ROM drives can also read audio CDs, sending the output to an external speaker or a sound card. You can use the MCI interface and the Media Player component to write applications that handle such a device. Basically, you need to set the `DeviceType` property to `dtCDAudio`, making sure no file is selected in the `FileName` property, and be ready with a CD player.

In fact, just by placing a Media Player component in a form, setting the above properties, and compiling and running the program, you end up with a fully functional audio CD player. When you start customizing the player, though, not everything is as simple as it seems at first glance. I’ve built an example using some more capabilities of this component and of Windows multimedia support related to audio CDs. The form of this program has a couple of buttons, some labels to show the current status, a timer, and a SpinEdit component you can use to choose a track from the disk.

The idea is to use the labels to inform the user of the number of tracks on a disk, the current track, the current position within a track, and the length of the track, monitoring the current situation using the timer.

In general, if you can, use the `tfTMSF` value (Track, Minute, Second, Frame) for the `TimeFormat` property of the Media Player component to access positional properties (such as `Position` and `Length`). Extracting the values is not too complex if you use the proper functions of the MmSystem unit, such as the following:

```
CurrentTrack := Mci_TMSF_Track (MediaPlayer1.Position);
```

Here are the two functions that compute the values for the whole disk and for the current track:

```
procedure TForm1.CheckDisk;
var
  NTracks, NLen: Integer;
begin
  NTracks := MediaPlayer1.Tracks;
  NLen := MediaPlayer1.Length;
  DiskLabel.Caption := Format (
    'Tracks: %.2d, Length:%.2d:%.2d', [NTracks,
    Mci_TMSF_Minute (NLen), Mci_TMSF_Second (NLen)]);
  SpinEdit1.MaxValue := NTracks;
end;

procedure TForm1.CheckPosition;
var
  CurrentTrack, CurrentPos, TrackLen: Integer;
begin
  CurrentPos := MediaPlayer1.Position;
  CurPosLabel.Caption := Format ('Position: %.2d:%.2d',
    [Mci_TMSF_Minute (CurrentPos),
    Mci_TMSF_Second (CurrentPos)]);
  CurrentTrack := Mci_TMSF_Track (CurrentPos);
  TrackLen := MediaPlayer1.TrackLength [CurrentTrack];
  TrackNumberLabel.Caption := Format (
    'Current track: %.2d, Length:%.2d:%.2d', [CurrentTrack,
    Mci_MSF_Minute (TrackLen), Mci_MSF_Second (TrackLen)]);
end;
```

The code seems complex only because of the many conversions it makes. Notice in particular that the length of the current track (stored in the `TrackLength` property) is not measured using the default format, as the online help suggests, but with the `MSF` (Minute Second Frame) format.

The global values for the disk are computed only at startup and when the New CD button is clicked:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.TimeFormat := tfTMSF;
  MediaPlayer1.Open;
  CheckDisk;
  CheckPosition;
end;

procedure TForm1.NewButtonClick(Sender: TObject);
begin
  CheckDisk;
  CheckPosition;
end;
```

The values for the current track and position are computed this way each time the timer interval elapses, by calling the `CheckPosition` method. This is far from perfect, because if you want to play an audio CD while using other programs, a timer accessing the Media Player information often slows down the system too much. Of course, this mainly depends on your hardware. Besides telling the user what is going on, the form has the Media Player component to allow the user to start and stop playing, change tracks, and so on. The operations on this component activate and halt the timer:

```
procedure TForm1.MediaPlayer1PostClick(
  Sender: TObject; Button: TMPBtnType);
begin
  if MediaPlayer1.Mode = mpPlaying then
    Timer1.Enabled := True
  else
    Timer1.Enabled := False;
  CheckPosition;
end;
```

You can also use the Go button to jump to the track selected in the SpinEdit component, where the `MaxValue` property is set by the `CheckDisk` method. Here is the code I've written:

```
procedure TForm1.GoButtonClick(Sender: TObject);
var
  Playing: Boolean;
begin
  Playing := (MediaPlayer1.Mode = mpPlaying);
  if Playing then
    MediaPlayer1.Stop;
  MediaPlayer1.Position :=
    MediaPlayer1.TrackPosition[SpinEdit1.Value];
  CheckPosition;
  if Playing then
    MediaPlayer1.Play;
end;
```

A good extension to this program would be to connect it to a CD database with the title of each CD you own and the title of each track. (I would have done that if it hadn't been for the time it would have taken to enter the title and track of each of my disks.) Remember, anyway, that a similar program is already available in Windows.

What's Next

In this chapter, we have seen how to add some audio and video capabilities to Delphi applications. We have seen how to add sound effects to respond to user actions. We have also seen examples of how to use the Media Player with sound files, video files, and an external device (an audio CD). With computers and CD-ROM players becoming faster every year, video is becoming an important feature of many applications. Don't underestimate this area of programming simply because you are writing *serious* business programs.

EPILOGUE

Delphi is a great programming environment. Now that you have learned about its core features, and seen the development of a number of programs, you might want to understand more about the language and fully master the internals of the VCL. This is where one of the books of the Mastering Delphi series I've written might help you. But you might also find more information in the other free e-books available on my web site.

As I mentioned in the Introduction, check the book page page, at

```
http://www.marcocantu.com/edelphi
```

for corrections and updates; the site has also links to other Delphi sites, documentation, simple wizards and components, and hosts also a newsgroup, which is the preferred way to report problems about this book (in the related section). It is possible to read the newsgroup also directly on the web from my site, and also sign-up to a mailing list in which I'll announce future edition of this and other volumes.

Again, as discussed in the introduction donations are welcome, also in the form of code examples, chapters for this and other books... but also offering cash, buying a printed book, or attending one of the Delphi classes I set up.