essential delphi ide 10.3

Sample content of a coming book introducing the Delphi IDE (and updated for version 10.3). The material is based (with permission) on classic editions of Mastering Delphi.

This ebook is copyright Marco Cantu 1995-2020. All rights reserved.

For more information refer to www.marcocantu.com

1: a form is a window

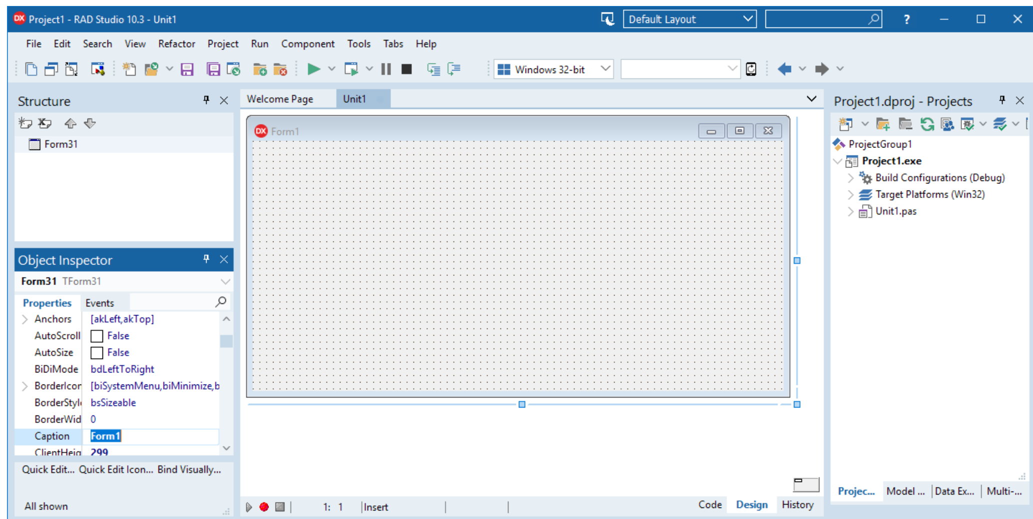
Let's start our exploration of Delphi by looking to the simplest scenario, building Windows applications using the VCL library. This is the easiest starting point, while most concept will also apply to mobile and multi-device development.

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 chapter 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.

> *Note:* 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. In the mobile world these relationships are a bit more intricate and platform-specific, but the broad concept still applies.

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 for a Windows application is one of the easiest operations in the system: You only need to create a new "Windows VCL Application" from the File | New menu or the Welcome page and Delphi will add to it a new, empty form, as you can see in the figure below. That's all there is to it.

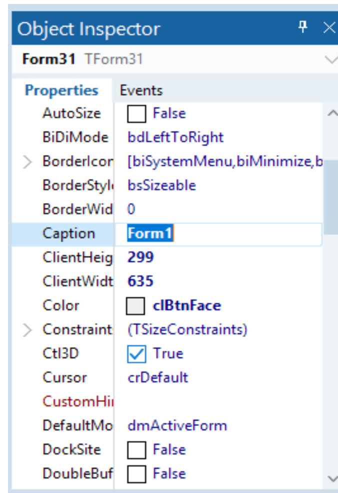


Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command or just pressing F9, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with no capabilities, but the default behavior of any Windows window.

Adding a Title

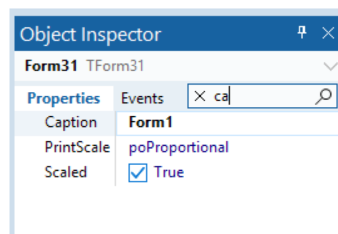
Before we run the application, let's make a quick change. The title of the form is Form1. For a user, the title of the main window stands for the name of the applica-

tion. 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 | Tools Windows | Object Inspector or pressing the F11 key):



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[1], so it's quite easy to find the ones you want to change, but you can also type the name of the property in the search box to get quickly to it:



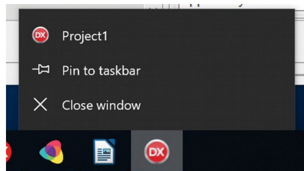
We can change the title of the form simply by changing the `Caption` property. While you type a new caption, you can see the title of the form change. If you type Hello, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. If you have not entered a

1 It is also possible to group properties in the Object Inspector by category, but this feature is seldom used by Delphi developers.

new caption, the new value of the `Name` property will be used for the `Caption` property, too.

> Tip: 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 menu in the Windows 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.

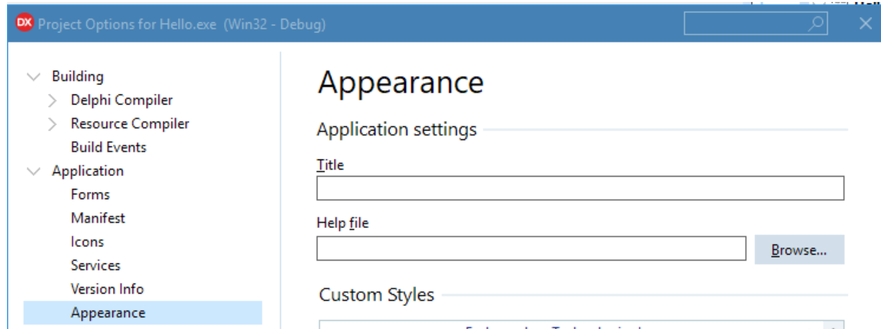
Saving the Form

Select the Save Project or Save Project As command from the File menu (or the matching toolbar button), and Delphi will first ask you to give a name to the source code file, or unit, 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 `HelloForm.pas`. 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 Form, just use Unit, or call every initial 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 | Appearance page of the Project Options dialog box (choose Project | Options):



As an alternative you can write a line of code to change the Title property of the Application global object, but that's for a later time.

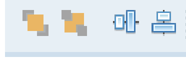
Using Components

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 Palette toolbox (by default on the right of the form designer, under the Project manager. If you choose the Button component from the Standard page of the Palette, for example, you can do any of the following four simple ways to place a component on a form:

- 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 copies of the component in the form using one of the above procedures. When you are done click on the Arrow button in the Palette toolbar to disable the selection.

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 | Toolbars | Position, a new section gets added to the Delphi IDE toolbar with positioning icons[2]:



This toolbox includes buttons to bring control in front of push the behind other controls, and center them in the form. A separate Align toolbar helps aligning controls one to the other. Using the last two buttons, 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 and the button won't be in the center anymore. So the button is only in the center of the form at start up. 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

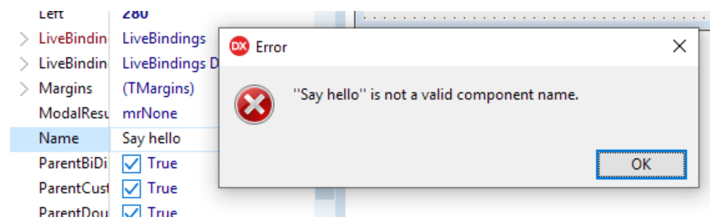
Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you 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.

> Note: 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

2 The Positioning toolbar (which was originally mixed with Align toolbar) is a rarely used feature these days. But I've decided to keep these steps and the example from the original Mastering Delphi book anyway.

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:

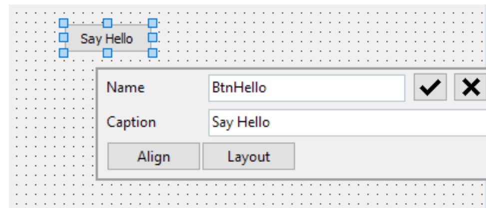


The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the `Name` property, you must follow the rules for naming an identifier in the Pascal language:

- An identifier is a sequence of letters, digits, or underscore characters of any length.
- 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.

Advanced Tip: You can use the `IsValidIdent` system function to check whether a given string is a valid identifier.

While in the sections above we've gone over manual steps, there is a faster way to set key properties of a component like its `Name` and `Caption`, and that is the use of the Quick Edit feature. Select the component in the form designer, right click on it, and pick the Quick Edit menu item. You'll see near the component a pane that allows you to quickly edit those two properties and buttons to expand panels for managing other common features:



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
28
end;
30
end.
```

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.

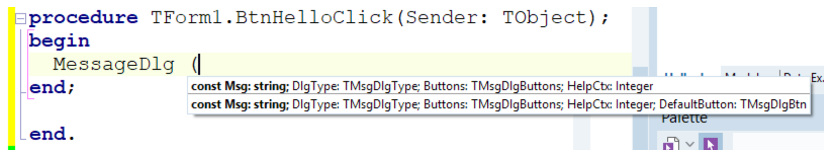
> Note: When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the begin and end 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, like `//`, 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. You can find many tutorials online if you need help or check the bibliography in the appendix of this book.

Of the code below, 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.



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):

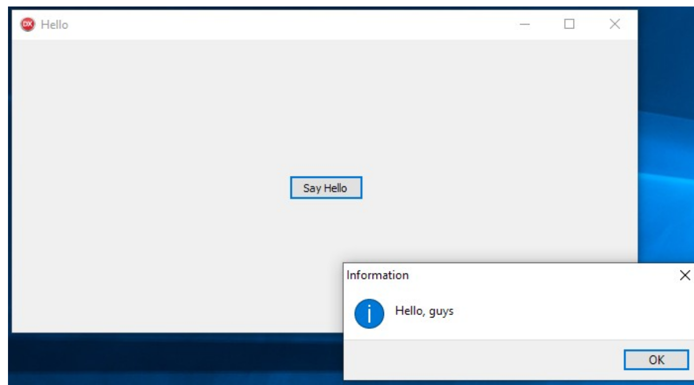
- The first parameter of the `MessageDlg` function is the string you want to display: the message.
- The second parameter is the type of message box. You can choose `mtWarning`, `mtError`, `mtInformation`, or `mtConfirmation`. For each type of message, the corresponding caption is used and a proper icon 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.

- 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.

> Note: Programmers unfamiliar with the Pascal language might be confused by the distinction between a function and a procedure. In Pascal and Delphi, there are two different keywords to define procedures and functions. The only difference between the two is that functions have a return value, while procedures are like "void functions" in C/C++ terms.

After you have written the line of code above, 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 Delphi compiler completely ignores new lines, white spaces, tab spaces, and similar formatting characters. Unlike other programming languages, 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 (.pas) describing the form

2: Compiles the project file (.dpr)

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

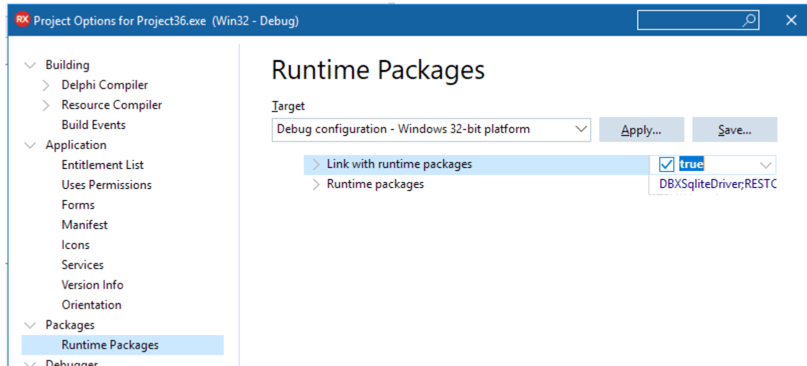
4: Runs the executable file, usually in debug mode

The executable file you obtained by default is a stand-alone executable program with no dependency on library file or a run-time library (as it happens for many other competing tools). Delphi allows you also to link the required libraries code into the executable file, but you can also specify the use of separate run-time packages, making the executable file much smaller but introducing a run-time dependency.

> Note: The fact that Delphi produces standalone executable files implies it does not require run-time compilation of a source code file (like it happens for JavaScript or Python), it doesn't need compilation of intermediate byte-code or IL representation (like C# or Java), and it doesn't even require an execution environment like the .NET run-time or the Java virtual machine. Delphi executable is just binary assembly of the given CPU and platform, desktop or mobile.

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 few hundred Kb (much more with debug information). 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 run-time packages:



Packages are dynamic link libraries containing Delphi components (the Visual Components Library, for example). 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 vclxxx.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).

> Note: The xxx in vclxxx.bpl stands for the specific version number, such as csl260.bpl for Delphi 10.3.x. Each major version of Delphi is incompatible with libraries for previous versions, and has a new set of run-time packages, with a different number.

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 executable files 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 than the equivalent code in interpreted or semi-compiled tools (although improvements in JIT – Just-In-Time compilation – tech-

nology has reduce the gap) and very fast to start as there is no initial compilation time to incur.

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.

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. 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 (the compiled DCU file, more on this later in the chapter).

> Note: To be honest the compilers based on the LLVM architecture (that is, most of the non-Windows compilers) are not as fast compiling and significantly slower when linking, as they use more standard techniques of the LLVM architecture and are less optimized.

Changing Properties at Run-Time

Let's return to the Hello application. We now want 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;
```

> Note: If you are new to the language, notice that Pascal and Delphi use 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 (or in its source code). Some of these run-time properties are defined as read-only, which means that you can access their value but you 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.

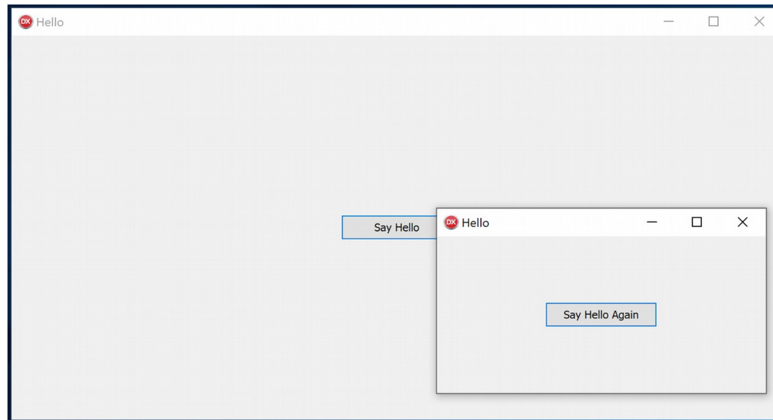
One solution is to change the border of the form to a thin frame, so that the form cannot be resized at run-time. Just move to the `BorderStyle` property of the form, and choose `bsSingle` instead of `bsSizeable` from the combo box.

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, of course. As you become more expert (and your applications become larger), the percentage of the time spent writing code will generally increase significantly.

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 (a method, to be precise) 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 form, dividing the height and the width of the internal area or client area of the form 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 or using Run Without Debugging from the Delphi IDE. By contrast, the Delphi environment can start only one copy of a program in debugging. When you use the Run button to start a program within Delphi, you execute it in the integrated debugger, and the IDE cannot debug two programs at the same time.

Note: You could possibly start two copies of the Delphi IDE and let each debug a different application, but this is a more advanced use case tha

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). 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

HelloForm.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 HelloForm.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
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.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 `HelloForm.pas`, the code begins with

```
unit HelloForm;
```

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).

> Note: FireMonkey multi-device applications use a very similar structure, only the textual definition of resources is saved in a file with the `.fmx` 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.

It is worth noting that there is a single file for the whole code of a form, not just small fragments for each of the event handlers. This is the complete code of the unit (something I'd generally avoid to list in the book, as it repeats a lot of boilerplate code):

```
unit HelloForm;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    BtnHello: TButton;
    procedure BtnHelloClick(Sender: TObject);
    procedure FormClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

uses
  System.UITypes;

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

procedure TForm1.FormClick(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.
```

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 text file with the definition of the configuration you create at design time with the form designer and Object Inspector.

> Note: In the early versions of Delphi the DFM file was a binary file. Now this is by default a text file converted to a binary resource during the compilation process. The binary version is what gets into the executable, because it is a more compact representation and a faster to process one. Whatever the format, if you load this file in the Delphi code editor, it will be converted into a textual description.
> In any case, you can determine if the DFM is text or binary for a new module by opening the IDE Tools | Options menu and selecting User Interface | Form Designer going over the Module creation options and using the check box *New forms as text*.

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 (again, in the book I'll generally include snippets of DFM files, but rarely a complete listing):

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Hello'
  ClientHeight = 299
  ClientWidth = 635
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
```

```
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  OnClick = FormClick
  OnResize = FormResize
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 261
    Top = 137
    Width = 113
    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`).

> 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 verse. 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
- You can manually edit the project file directly

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

```
program Hello;

uses
  Vcl.Forms,
  HelloForm in 'HelloForm.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

You can see this file with the Project | View Source menu command (historically it was View | Project Source). As an alternative, you can select the project node in the Project manager and use the View Source option of the local menu.

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.

In the next chapter we'll start exploring the Delphi IDE is a more systematic way, going over the various features it has. Following chapter will delve into very specific areas.

The example 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.

2: highlights of the delphi ide

In a visual programming tool such as Delphi, the role of the environment is certainly important, and the various tools help you get work done faster. After the introduction in the last chapter, this second part offers a deeper overview of the IDE and its features. Now in some cases the topics just introduce deeper coverage in following chapters, while in others there isn't much more to say.

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 two of them, with some variations:

- The Professional edition is aimed at professional developers building stand-alone applications or simple database ones (the FireDAC data access library is included, with limited client/server support). The Professional package has limitations in multi-tier development, but offers the full set of controls for UI development on desktop and mobile.
- The Community Edition (CE) has the same features of the Professional edition, from a technical point of view, but comes with a limited license:
  - You can use it only if you or your company makes less than 5,000 USD/year in revenues (covering students, hobbyist, retired people, start-ups, and more)
  - You can only install a maximum of 5 copies on your local network (notice that educational institution looking to install many copies on a lab can use the Academic versions, which are free or have a nominal fee)
- The Enterprise edition is aimed at developers building client/server and multi-tier applications. It includes all FireDAC drivers for most Enterprise level relational databases, DataSnap and RAD Server multi-tier architectures, and support for the Linux target.

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. You can also install a large number of different IDE add-ins, a few of which are made available directly by Embarcadero and I'll mention in the book.

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.

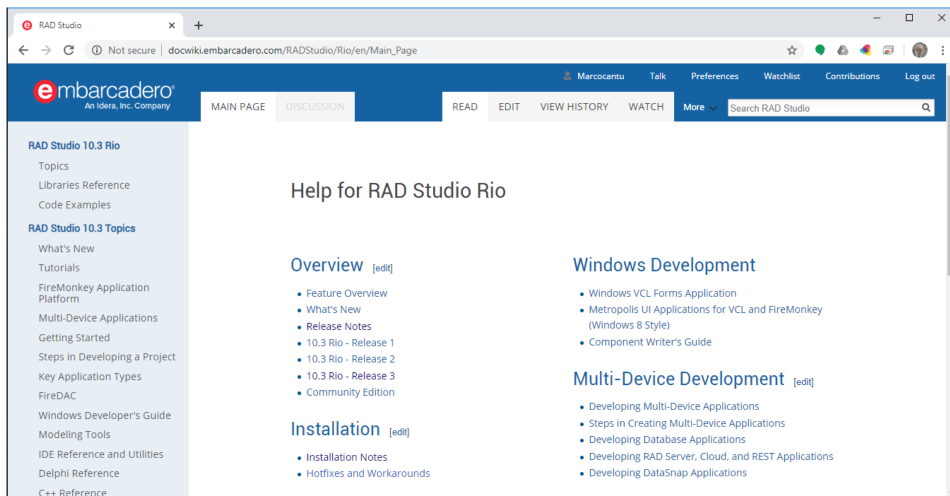
When you press F1, Delphi doesn't search for an exact match in the Help Search list. Instead, it tries to understand what you are asking. For example, if you press F1 when the text cursor is on the name of the `Button1` component in the source code, the Delphi Help system automatically opens the description 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.

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.

As an alternative you can refer to the online product documentation, powered by a Wiki engine, at

```
http://docwiki.embarcadero.com/RADStudio/en/Main_Page
```

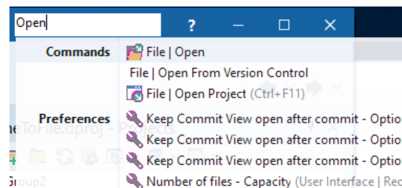


Tip: The online version of the documentation on DocWiki gets updated more frequently than the version installed in the product.

Delphi Menus and Commands

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

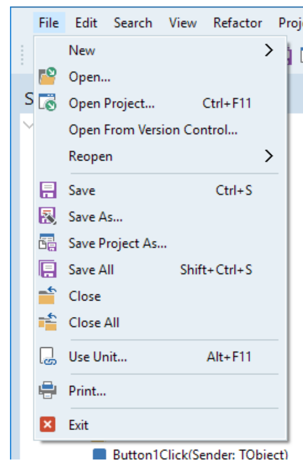
- Use the main menu
- Use the (customizable) toolbar
- Use one of the local menus activated by pressing the right mouse button in the various panes
- Use IDE Insight (the search box in the title bar, covered in more detail in section xxx) and type the name of the command you are looking for, like:



The Delphi menus offer many commands and the IDE is very rich of features. While it might look to be boring, in this section I'll go over each sub-menu of the main menu and the various items as this gives me a very good way to offer an overview of the features of the IDE. In the following sections, I'll present some suggestions on the use of some of the menu commands. In same cases I'll just mention a feature which is detailed in a later chapter and refer to the deeper coverage elsewhere.

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 specific types of projects added and removed over time. Still, this menu contains commands that operate on projects and commands that operate on source code files and offers options for opening files, creating new ones, saving and closing.



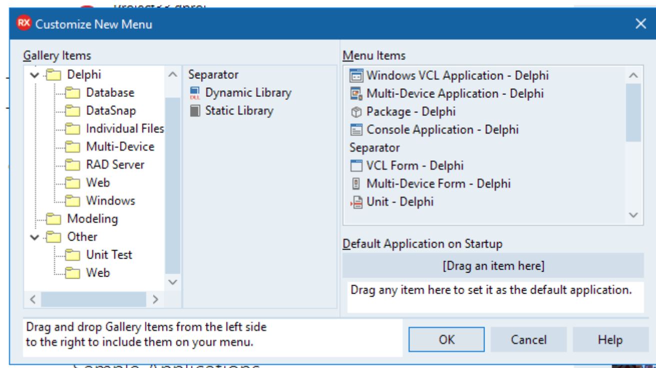
The File | New Sub-Menu

The File | New sub-menu offers different options for common operations and the ability to open the File | New | Other dialog, also known as Object Repository. The basic options are those that tend to change over time. Currently (in Delphi 10.3) the menu offers by default the ability to create the following new items:

- *Windows VCL Application* creates a standard, empty project for the Windows platform only, based on the classic VCL library
- *Multi-Device Application* creates a FireMonkey application for desktop and mobile platforms. You can select one of a few predefined structures (with headers, footers, tabs, and more) or start with an empty form, or *Blank Application*.
- *Package* creates a components package or a package hosting IDE extensions or other features. Packages are a slightly more advanced topics than we can really cover in this book.
- *Console Application* creates a text-based console app you can use for different operating systems (including Linux, if you have the Enterprise version). A console app can just use direct interaction with the use via standard text input and output, and is often used for writing test or small utilities. A console application starts with basic, skeleton code.
- *VCL Form*, *Multi-Device Form*, and *Unit* create a new standard alone Pascal source code file or add a new one to the current project (if one is active). If a project is open, however, only the compatible elements are visible (that is, if

you are working on a VCL project you won't see *Multi-Device Form* menu item). In each case the new Pascal source code file (unit) will have a standard basic structure for forms or will be empty if you pick a plan unit.

- *Other* opens the File | New | Other dialog, or Object Repository. See section xxx for more information.
- *Customize* allows you to change the entries of the File | New menu adding new entries you use often or removing the current ones. The dialog to customize the File | New menu looks like the following (but actual content depends on your edition of RAD Studio or Delphi:

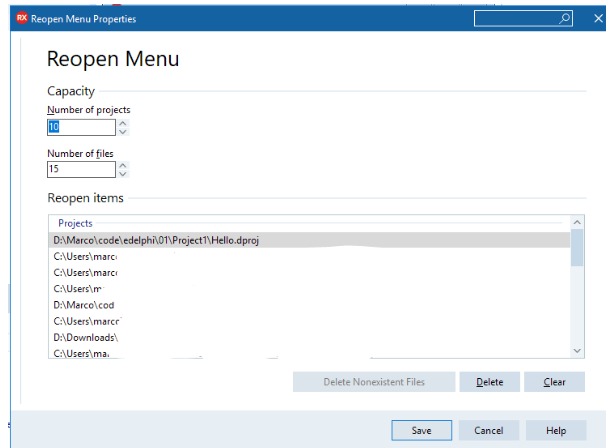


The File | Open Commands

Let's now get back to the main File menu. There are 4 open commands:

- *Open* can be used to open any file, including units, projects, project groups, but also plan text files, INI files, configuration files, HTML files or any other file a text editor can handle. Opening a new file generally doesn't affect the current open project or open files. Notice that there are different commands for adding an existing unit to the current project.
- *Open Project* can be used to open an exiting project, replacing the currently open project if any. Again, there is an alternative option which is add a new project to the existing project group, keeping both (or many) projects open ant the same time.
- Open From Version Control allows you to open a project from a remote repository in a Version Control System like Subversion, Git or Mercurial. There is more information on this topic in Chapter ***.

- Reopen allows you to open a recently closed project or file. When you select File | Reopen you see a list of recently closed files and projects. You can also customize how many of these files are kept in each group and do some cleanup in the list by using the File | Reopen | Properties menu, which leads to the following dialog box:



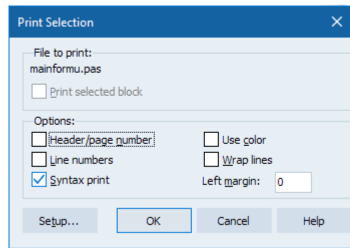
The Other File Menu Commands

Saving files and projects is quite straightforward, with the menu commands *Save*, *Save As*, *Save Project As*, *Save All*. Notice that if you save a project, Delphi will prompt you to save the existing units first, and give a name to any newly created one.

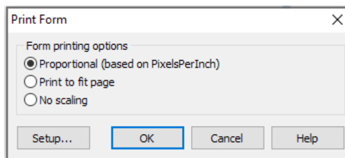
The Close and Close All commands are self-explicative. Notice, however, the local menu of editor tabs offers additional closing operations, like closing all files save for the current one or all those to the left and right (considering that position generally depends on the opening sequence).

The following File menu command, *Use Unit*, offers the ability to add a reference from the current unit to another unit in the project (with a `uses` statement). The interesting point here is that this operation will let you add references to components in the other unit at design time in the IDE, for example connecting a visual component to a data source in another unit, like a Data Module.

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, however, the printer will produce the graphical representation of the form, offering these options:



Finally, there is the *Exit* menu item, which prompts you for saving any open file or project ans shuts down the IDE.

Tip: It is worth noting that it is not fully recommended to keep a very complex software like Delphi running for many days in a row. Shutting it down from time to time cleaning up all memory and resources is a good recommendation.