essential delphi

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

For more information refer to www.marcocantu.com

1: a form is a window

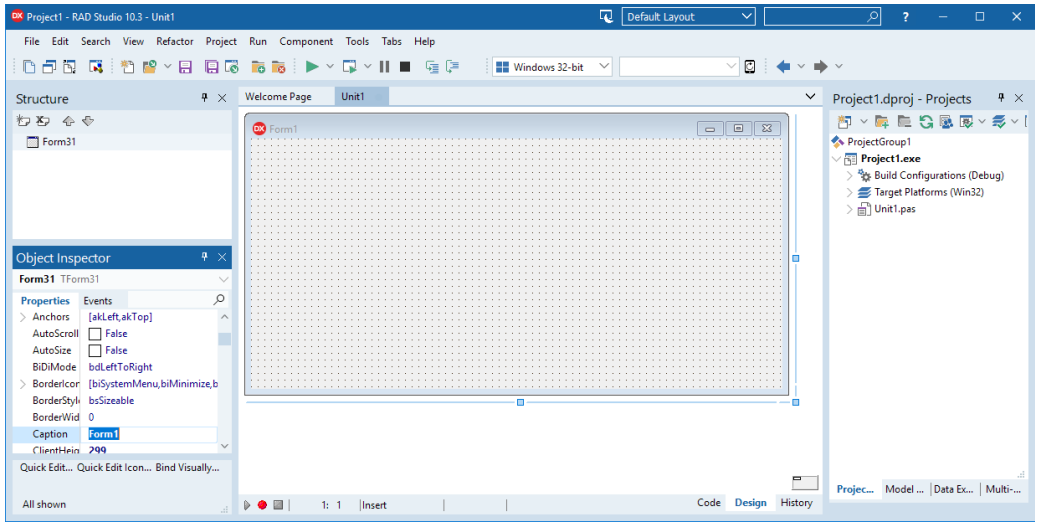
Let's start our exploration of Delphi by looking to the simplest scenario, building Windows applications using the VCL library. This is the easiest starting point, while most concept will also apply to mobile and multi-device development.

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.

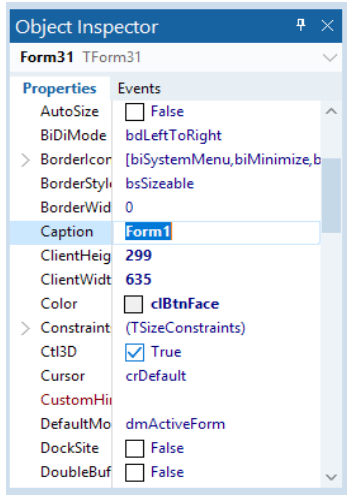


Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command or just pressing F9, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with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

Adding a Title

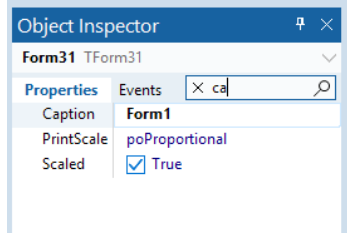
Before we run the application, let's make a quick change. The title of the form is Form1. For a user, the title of the main window stands for the name of the applica-

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:



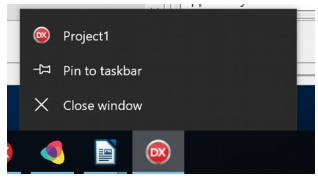
We can change the title of the form simply by changing the `Caption` property. While you type a new caption, you can see the title of the form change. If you type Hello, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. 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.

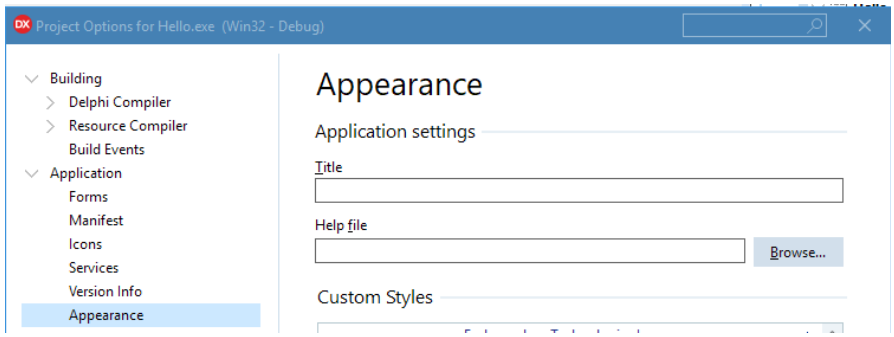
Saving the Form

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.

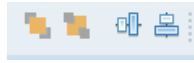
Using Components

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

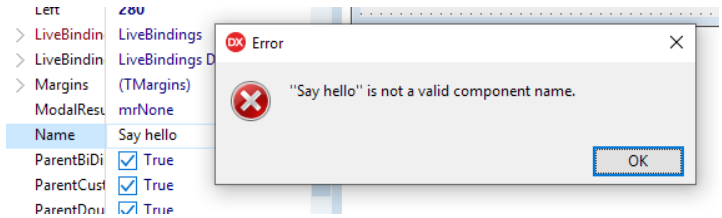
Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you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

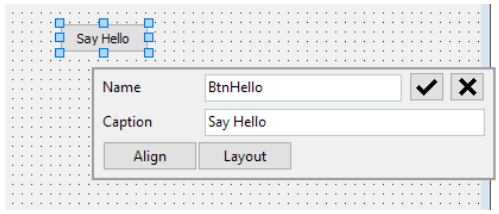


The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the `Name` property, you must follow the rules for naming an identifier in the Pascal language:

- 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
  |
end;

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.

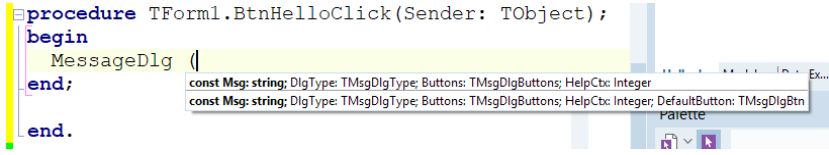
> Note: When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the begin and end 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):

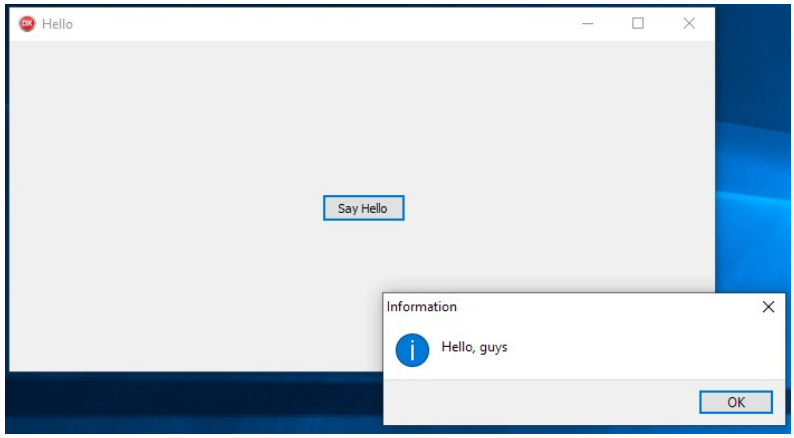
- The first parameter of the `MessageDlg` function is the string you want to display: the message.
- The second parameter is the type of message box. You can choose `mtWarning`, `mtError`, `mtInformation`, or `mtConfirmation`. For each type of message, the corresponding caption is used and a proper icon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