Your Very First Visual Basic Program

Visual Basic lets you build a complete and functional Windows application by dropping a bunch of controls on a form and writing some code that executes when something happens to those controls or to the form itself. For instance, you can write code that executes when a form loads or unloads or when the user resizes it. Likewise, you can write code that executes when the user clicks on a control or types while the control has the input focus.

This programming paradigm is also known as *event-driven programming* because your application is made up of several event procedures executed in an order that's dependent on what happens at run time. The order of execution can't, in general, be foreseen when the program is under construction. This programming model contrasts with the procedural approach, which was dominant in the old days.

This section offers a quick review of the event-driven model and uses a sample application as a context for introducing Visual Basic's intrinsic controls, with their properties, methods, and events. This sample application, a very simple one, queries the user for the lengths of the two sides of a rectangle, evaluates its perimeter and area, and displays the results to the user. Like all lengthy code examples and programs illustrated in this book, this application is included on the companion CD.

Adding Controls to a Form

We're ready to get practical. Launch the Visual Basic IDE, and select a Standard EXE project. You should have a blank form near the center of the work area. More accurately, you have a *form designer,* which you use to define the appearance of the main window of your application. You can also create other forms, if you need them, and you can create other objects as well, using different designers (the UserControl and UserDocument designers, for example). Other chapters of this book are devoted to such designers.

One of the greatest strengths of the Visual Basic language is that programmers can design an application and then test it without leaving the environment. But you should be aware that designing and testing a program are two completely different tasks. At *design time,* you create your forms and other visible objects, set their properties, and write code in their event procedures. Conversely, at *run time* you monitor the effects of your programming efforts: What you see on your screen is, more or less, what your end users will see. At run time, you can't invoke the form designer, and you have only a limited ability to modify the code you have written at design time. For instance, you can modify existing statements and add new ones, but you can't add new procedures, forms, or controls. On the other hand, at run time you can use some diagnostic tools that aren't available at design time because they would make no sense in that context (for example, the Locals, the Watches, and the Call Stack windows).

To create one or more controls on a form's surface, you select the control type that you want from the Toolbox window, click on the form, and drag the mouse cursor until the control has the size and shape you want. (Not all controls are resizable. Some, such as the Timer control, will allow you to drag but will return to their original size and shape when you release the mouse button.) Alternatively, you can place a control on the form's surface by double-clicking its icon in the Toolbox: this action creates a control in the center of the form. Regardless of the method you follow, you can then move and resize the control on the form using the mouse.

> **TIP**
>
> If you need to create multiple controls of the same type, you can follow this three-step procedure: First, click on the control's icon on the Toolbox window while you keep the Ctrl key pressed. Next, draw multiple controls by clicking the left button on the form's surface and then dragging the cursor. Finally, when you're finished creating controls, press the Escape key or click the Pointer icon in the upper left corner of the Toolbox.

To complete our Rectangle sample application, we need four TextBox controls—two for entering the rectangle's width and height and two for showing the resulting perimeter and area, as shown in Figure 1-8. Even if they aren't strictly required from an operational point of view, we also need four Label controls for clarifying the purpose of each TextBox control. Finally we add a CommandButton control named *Evaluate* that starts the computation and shows the results.

Place these controls on the form, and then move and resize them as depicted in Figure 1-8. Don't worry too much if

the controls aren't perfectly aligned because you can later move and resize them using the mouse or using the commands in the Format menu.

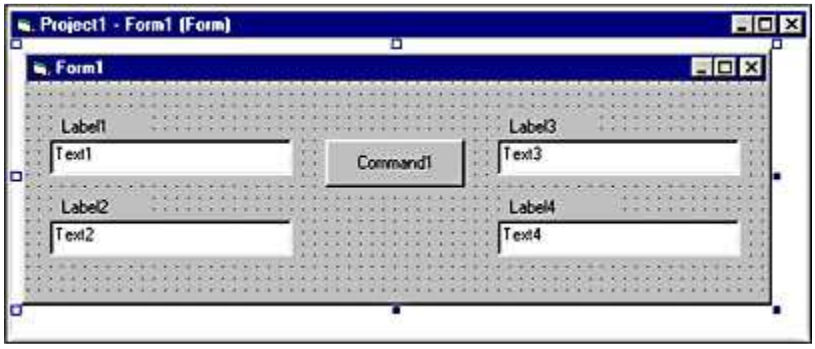

Figure 1-8 *The Rectangle Demo form at design time, soon after the placement of its controls.*

Setting Properties of Controls

Each control is characterized by a set of properties that define its behavior and appearance. For instance, Label controls expose a *Caption* property that corresponds to the character string displayed on the control itself, and a *BorderStyle* property that affects the appearance of a border around the label. The TextBox control's most important property is *Text*, which corresponds to the string of characters that appears within the control itself and that can be edited by the user.

In all cases, you can modify one or more properties of a control by selecting the control in the form designer and then pressing F4 to show the Properties window. You can scroll through the contents of the Properties window until the property you're interested in becomes visible. You can then select it and enter a new value.

Using this procedure, you can modify the *Caption* property of all four Label controls to *&Width*, *&Height*, *&Perimeter*, and *&Area*, respectively. You will note that the ampersand character doesn't appear on the control and that its effect is to underline the character that follows it. This operation actually creates a *hot key* and associates it with the control. When a control is associated with a hot key, the user can quickly move the focus to the control by pressing an Alt+*x* key combination, as you normally do within most Windows applications. Notice that only controls exposing a *Caption* property can be associated with a hot key. Such controls include the Label, Frame, CommandButton, OptionButton, and CheckBox.

> **TIP**
>
> There is one handy but undocumented technique for quickly selecting a given property of a control. You just have to select the control on the form and press the Ctrl+Shift+*x* key, where *x* is the first letter in the property's name. For instance, select a Label control, and then press Ctrl+Shift+C to display the Properties window and select the *Caption* property in one operation. Pressing the Ctrl+Shift+C key again moves the focus to the next property whose name begins with the C character, and so on in a cyclic fashion.

Notice that once you have selected the *Caption* property for the first Label control, it stays selected when you then click on other controls. You can take advantage of this mechanism to change the *Caption* property of the CommandButton control to *&Evaluate* and the *Caption* property of the Form itself to *Rectangle Demo,* without having to select the *Caption* item in the Properties window each time. Note that ampersand characters within a form's caption don't have any special meaning.

As an exercise, let's change the font attributes used for the controls, which you do through the *Font* property. While you can perform this action on a control-by-control basis, it's much easier to select the group of controls that you want to affect and then modify their properties in a single operation. To select multiple controls, you can click on each one of them while you press either the Shift or the Ctrl key, or you can drag an imaginary rectangle around them. (This technique is also called *lassoing* the controls.)

TIP

A quick way to select all the controls on a form is to click anywhere on the form and press the Ctrl+A key combination. After selecting all controls, you can deselect a few of them by clicking on them while pressing the Shift or Ctrl key. Note that this shortcut doesn't select controls that are contained in other controls.

When you select a group of controls and then press the F4 key, the Properties window displays only the properties that are common to all the selected controls. The only properties that are exposed by any control are *Left*, *Top*, *Width*, and *Height*. If you select a group of controls that display a string of characters, such as the TextBox, Label, and CommandButton controls in our Rectangle example, the *Font* property is also available and can therefore be selected. When you double-click on the *Font* item in the Properties window, a Font dialog box appears. Let's select a Tahoma font and set its size to 11 points.

TIP

If you want to copy a number of properties from one control to one or more other controls, you can select the control you want to copy from, press Shift and select the other controls, press F4 to show the Properties window, and triple-click the name of the property you want to copy. Note that you must click the name of the property on the left, not the value cell on the right. The values of the properties on which you triple-click are copied from the source controls to all the other selected controls. This technique doesn't work with all the items in the Properties window.

Finally we must clear the *Text* property of each of the four TextBox controls so that the end user will find them empty when the program begins its execution. Oddly, when you select two or more TextBox controls, the *Text* property doesn't appear in the Properties window. Therefore, you must set the *Text* property to an empty string for each individual TextBox control on the form. To be honest, I don't know why this property is an exception to the rule stated earlier. The result of all these operations is shown in Figure 1-9.

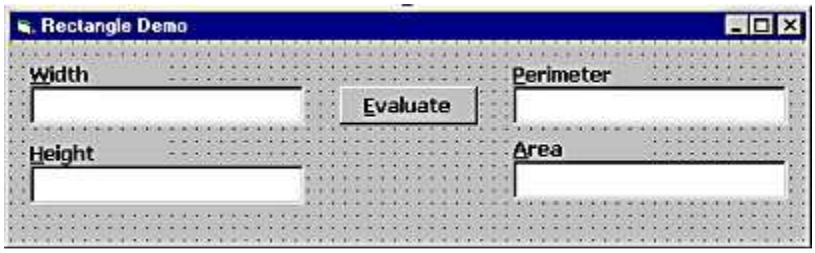

Figure 1-9. *The Rectangle Demo form at design time, after setting the controls' properties.*

TIP

When a control is created from the Toolbox, its *Font* property reflects the font of the parent form. For this reason, you can often avoid individual font settings by changing the form's *Font* property before placing any controls on the form itself.

Naming Controls

One property that every control has and that's very important to Visual Basic programmers is the *Name* property. This is the string of characters that identifies the control in code. This property can't be an empty string, and you can't have two or more controls on a form with the same name. The special nature of this property is indirectly confirmed by the fact that it appears as (Name) in the Properties window, where the initial parenthesis serves to move it to the beginning of the property list.

When you create a control, Visual Basic assigns it a default name. For example, the first TextBox control that you place on the form is named *Text1*, the second one is named *Text2*, and so forth. Similarly, the first Label control is named *Label1*, and the first CommandButton control is named *Command1*. This default naming scheme frees you from having to invent a new, unique name each time you create a control. Notice that the *Caption* property of Label and CommandButton controls, as well as the *Text* property of TextBox controls, initially reflect the control's *Name* property, but the two properties are independent of each other. In fact, you have just modified the *Caption* and *Text* properties of the controls in the Rectangle Demo form without affecting their *Name* properties.

Because the *Name* property identifies the control in code, it's a good habit to modify it so that it conveys the meaning of the control itself. This is as important as selecting meaningful names for your variables. In a sense, most controls on a form are special variables whose contents are entered directly by the user.

Microsoft suggests that you always use the same three-letter prefix for all the controls of a given class. The control classes and their recommended prefixes are shown in Table 1-1.

Table 1-1. *Standard three-letter prefixes for forms and all intrinsic controls.*

Control Class	***Prefix***	***Control Class***	***Prefix***
CommandButton	cmd	Data	dat
TextBox	txt	HScrollBar	hsb
Label	lbl	VScrollBar	vsb
PictureBox	pic	DriveListBox	drv
OptionButton	opt	DirListBox	dir
CheckBox	chk	FileListBox	fil
ComboBox	cbo	Line	lin
ListBox	lst	Shape	shp
Timer	tmr	OLE	ole
Frame	fra	Form	frm

For instance, you should prefix the name of a TextBox control with txt, the name of a Label control with lbl, and the name of a CommandButton control with cmd. Forms should also follow this convention, and the name of a form should be prefixed with the *frm* string. This convention makes a lot of sense because it lets you deduce both the control's type and meaning from its name. This book sticks to this naming convention, especially for more complex examples when code readability is at stake.

In our example, we will rename the Text1 through Text4 controls as txtWidth, txtHeight, txtPerimeter, and txtArea respectively. The Command1 control will be renamed cmdEvaluate, and the four Label1 through Label4 controls will be renamed lblWidth, lblHeight, lblPerimeter, and lblArea, respectively. However, please note that Label controls are seldom referred to in code, so in most cases you can leave their names unmodified without affecting the code's readability.

Moving and Resizing Controls

You probably won't be able to place your controls on the form in the right position on your first attempt. Most likely, you will try several layouts until you are satisfied with the overall appearance of the form. Fortunately, the IDE offers you many ways to modify the position and size of your controls without much effort.

- Select one or more controls, and move them as a single entity using the mouse.
- Move one or more controls with arrow keys while you press the Ctrl key. The steps along the x- and y-axes are determined by the Grid Units settings. You can view and modify these settings using the General tab of the Options dialog box from the Tools menu.

- Resize the selected control(s) by using the arrow keys while you press the Shift key. You can also resize a control by dragging one of the blue handles surrounding it when it is selected. Like the move operation, the resize step depends on the Grid Units settings.

- Center a control or a group of controls on the form, either horizontally or vertically, using the Center In Form submenu of the Format menu.

- Align a group of controls with respect to another control using the commands in the Align submenu of the Format menu. The control used as a reference in the aligning process is the one that was selected last (that is, the one with blue handles around it).

- Resize a group of controls by selecting them and invoking a command in the Make Same Size submenu of the Format menu. All selected controls will be resized to reflect the size of the control that was selected last.

- You can align or resize a group of controls by selecting them, pressing F4 to display the Properties window, and then manually modifying the *Left*, *Top*, *Width*, or *Height* properties. This procedure is useful when you know the absolute position or size of the controls.

TIP

A ComboBox control is peculiar in that its height is determined by the system and depends on its *Font* property. Therefore, if you have ComboBox and single-line TextBox controls on the same form, you should use either one of the last two techniques that I just described to resize the TextBox controls to reflect the height of the ComboBox control (s) placed on the form. This will give your form a consistent look.

Setting the Tab Order

Windows standards dictate that the user can press the Tab key to visit all the fields in a window in the logical order. Such a sequence is known as the *Tab order* sequence. In Visual Basic, you set the correct Tab order sequence by assigning a proper value to the *TabIndex* property for all the controls that can receive the input focus, starting with 0 for the control that should receive the input focus when the form appears and assigning increasing values for all the others. In our Rectangle sample application, this means assigning 0 to the txtWidth control's *TabIndex* property, 1 to the txtHeight control's *TabIndex* property, and so on.

But wait, there's more to know about the Tab order setting. Even if Label controls never get the focus themselves, they expose a *TabIndex* property. Why?

As I mentioned previously, TextBox controls—or more to the point, controls that don't expose a *Caption* property—can't be directly associated with a hot key. This means that you can't use an Alt+*x* key combination to activate them. In our Rectangle example, we overcome this limitation by placing Label controls above each individual TextBox control. Unfortunately, placing a Label control near another control doesn't automatically provide it with hot key capabilities. To have a Label control lend its hot key to another control on the form, you must assign the Label's *TabIndex* property a value that is 1 less than the value of the other control's *TabIndex* property.

In our Rectangle sample application, this means assigning the TabIndex property as follows: 0 to lblWidth, 1 to txtWidth, 2 to lblHeight, 3 to txtHeight, 4 to cmdEvaluate, 5 to lblPerimeter, 6 to txtPerimeter, 7 to lblArea, and 8 to txtArea.

It's immediately apparent that when you have forms with tens or even hundreds of controls, correctly setting the *TabIndex* property for each one of them is a nuisance. For this reason, a number of third-party commercial or shareware vendors have developed special add-ins that permit you to solve this task in a visual manner, for example by clicking on each control, or in a semiautomatic manner by analyzing the relative position of all controls on the form. While these add-ins are real lifesavers, here's a trick well known among Visual Basic programmers that solves the problem with relatively little effort:

1. Select the *last* control in your planned Tab order.

2. Press the Ctrl+Shift+T key combination to activate the Properties window. For most controls, this combination

selects the *TabIndex* properties; for others, you might need to press it more than once.

3. Press the 0 key, thus assigning a 0 to the *TabIndex* property of the selected control.

4. Click on the next to last control in the Tab order, and press the 0 key again; this assigns a 0 to the *TabIndex* property of the current control and 1 to the *TabIndex* property of the last control. This occurs because Visual Basic prevents you from using the same *TabIndex* value for two or more controls on the same form.

5. Repeat step 4, working backward in the Tab order sequence and pressing the 0 key after selecting each control. When you reach the first control in the sequence, the *TabIndex* property for all the controls on the form will be set correctly.

> **TIP**
>
> Visual Basic 5 and 6 also come with an add-in that permits you to arrange the *TabIndex* property for all the controls on the current form. This add-in is provided in source code format, in the TabOrder.vbp project located in the Samples\CompTool\AddIns subdirectory. To use this add-in, you must compile and install it manually. This tool lets you save a lot of time when arranging the Tab order for forms with many controls.

Now that we have completed our project, we'll save it. Choose Save Project from the File menu, or click the floppy disk icon. Visual Basic will ask you for the name of the form file, and again for the name of the project file; type *Rectangle* for both. You'll see that you now have two new files, Rectangle.frm and Rectangle.vbp.

Adding Code

Up to this point, you have created and refined the user interface of your program and created an application that in principle can be run. (Press F5 and run it to convince yourself that it indeed works.) But you don't have a useful application yet. To turn your pretty but useless program into your first working application, you need to add some code. More precisely, you have to add some code in the *Click* event of the cmdEvaluate control. This event fires when the user clicks on the Evaluate button or presses its associated hot key (the Alt+E key combination, in this case).

To write code within the *Click* event, you just select the cmdEvaluate control and then press the F7 key, or right-click on it and then invoke the View Code command from the pop-up menu. Or you simply double-click on the control using the left mouse button. In all cases, the code editor window appears, with the flashing cursor located between the following two lines of code:

```
Private Sub cmdEvaluate_Click()

End Sub
```

Visual Basic has prepared the template of the *Click* event procedure for you, and you have to add one or more lines of code between the *Sub* and *End Sub* statements. In this simple program, you need to extract the values stored in the txtWidth and txtHeight controls, use them to compute the rectangle's perimeter and area, and assign the results to the txtPerimeter and txtArea controls respectively:

```
Private Sub cmdEvaluate_Click()
    ' Declare two floating point variables.
    Dim reWidth As Double, reHeight As Double
    ' Extract values from input TextBox controls.
    reWidth = CDbl(txtWidth.Text)
    reHeight = CDbl(txtHeight.Text)
    ' Evaluate results and assign to output text boxes.
    txtPerimeter.Text = CStr((reWidth + reHeight) * 2)
    txtArea.Text = CStr(reWidth * reHeight)
End Sub
```

TIP

Many developers, especially those with prior experience in the QuickBasic language, are accustomed to extracting numeric values from character strings using the *Val* function. The *CDbl* or *CSng* conversion functions are better choices in most cases, however, because they're *locale-aware* and correctly interpret the number in those countries where the decimal separator is the comma instead of the period. Even more important, the *CDbl* or *CSng* functions conveniently skip over separator characters and currency symbols (as in $1,234), whereas the *Val* function doesn't.

Note that you should always use the *Dim* statement to declare the variables you are going to use so that you can specify for them the most suitable data type. If you don't do that, Visual Basic will default them to the Variant data type. While this would be OK for this sample program, for most occasions you can make better and faster applications if you use variables of a more specific type. Moreover, you should add an *Option Explicit* statement at the very beginning of the code module so that Visual Basic will automatically trap any attempt to use a variable that isn't declared anywhere in the program code. By this single action, you'll avoid a lot of problems later in the development phase.

Running and Debugging the Program

You're finally ready to run this sample program. You can start its execution in several ways: By invoking the Start command from the Run menu, by clicking the corresponding icon on the toolbar, or by pressing the F5 key. In all cases, you'll see the form designer disappear and be replaced (but not necessarily in the same position on the screen) by the real form. You can enter any value in the leftmost TextBox controls and then click on the Evaluate button (or press the Alt+E key combination) to see the calculated perimeter and area in the rightmost controls. When you're finished, end the program by closing its main (and only) form.

CAUTION

You can also stop any Visual Basic program running in the environment by invoking the End command from the Run menu, but in general this isn't a good approach because it prevents a few form-related events—namely the *QueryUnload* and the *Unload* events—from firing. In some cases, these event procedures contain the so-called *clean-up code,* for example, statements that close a database or delete a temporary file. If you abruptly stop the execution of a program, you're actually preventing the execution of this code. As a general rule, use the End command only if strictly necessary.

This program is so simple that you hardly need to test and debug it. Of course, this wouldn't be true for any real-world application. Virtually all programs need to be tested and debugged, which is probably the most delicate (and often tedious) part of a programmer's job. Visual Basic can't save you from this nuisance, but at least it offers so many tools that you can often complete it very quickly.

To see some Visual Basic debugging tools in action, place a breakpoint on the first line of the *Click* event procedure while the program is in design mode. You can set a breakpoint by moving the text cursor to the appropriate line and then invoking the Toggle Breakpoint command from the Debug menu or pressing the F9 shortcut key. You can also set and delete breakpoints by left-clicking on the gray vertical strip that runs near the left border of the code editor window. In all cases, the line on which the breakpoint is set will be highlighted in red.

After setting the breakpoint at the beginning of the *Click* event procedure, press F5 to run the program once again, enter some values in the Width and Height fields, and then click on the Evaluate button. You'll see the Visual Basic environment enter break mode, and you are free to perform several actions that let you better understand what's actually going on:

- Press F8 to execute the program one statement at a time. The Visual Basic instruction that's going to be executed next—that is, the current statement —is highlighted in yellow.
- Show the value of an expression by highlighting it in the code window and then pressing F9 (or selecting the Quick Watch command from the Debug menu). You can also add the selected expression to the list of values

displayed in the Watch window, as you can see in Figure 1-10.

- An alternative way to show the value of a variable or a property is to move the mouse cursor over it in the code window; after a couple of seconds, a yellow *data tip* containing the corresponding value appears.

- Evaluate any expression by clicking on the Immediate window and typing *?* or *Print* followed by the expression. This is necessary when you need to evaluate the value of an expression that doesn't appear in the code window.

- You can view the values of all the local variables (but not expressions) by selecting the Locals command from the View menu. This command is particularly useful when you need to monitor the value of many local variables and you don't want to set up a watching expression for each one.

- You can affect the execution flow by placing the text cursor on the statement that you want to execute next and then selecting the Set Next Statement command from the Debug menu. Or you can press the Ctrl+F9 key combination. You need this technique to skip over a piece of code that you don't want to execute or to reexecute a given block of lines without restarting the program.

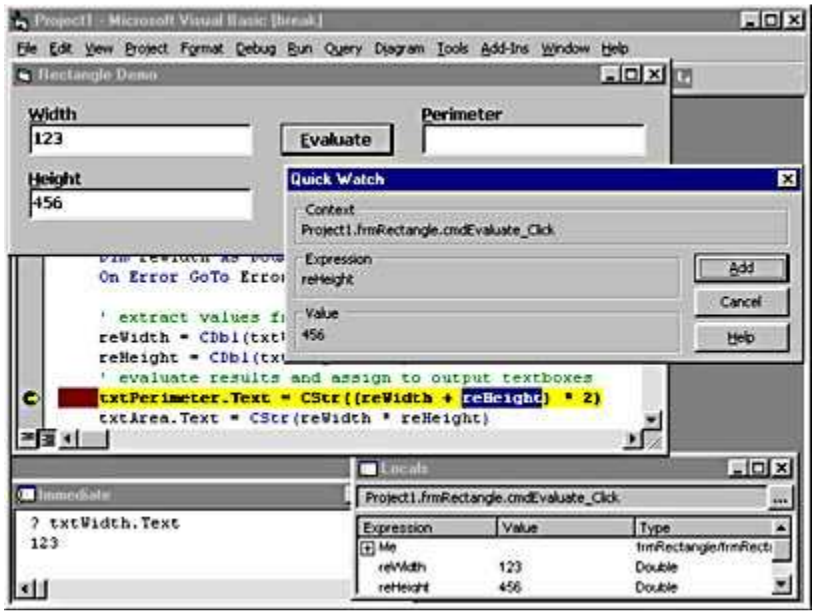

Figure 1-10. *The Rectangle Demo program in break mode, with several debug tools activated.*

Refining the Sample Program

Our first Visual Basic project, Rectangle.vbp, is just a sample program, but this is no excuse not to refine it and turn it into a complete and robust, albeit trivial, application.

The first type of refinement is very simple. Because the txtPerimeter and txtArea controls are used to show the results of the computation, it doesn't make sense to make their contents editable by the user. You can make them read-only fields by setting their *Locked* property to True. (A suggestion: select the two controls, press F4, and modify the property just once.) Some programmers prefer to use Label controls to display result values on a form, but using read-only TextBox controls has an advantage: The end user can copy their contents to the clipboard and paste those contents into another application.

A second refinement is geared toward increasing the application's consistency and usability. Let's suppose that your user uses the Rectangle program to determine the perimeter and area of a rectangle, takes note of the results, and then enters a new width or a new height (or both). Unfortunately, an instant before your user clicks on the Evaluate button the phone rings, engaging the user in a long conversation. When he or she hangs up, the form shows a plausible, though incorrect, result. How can you be sure that those values won't be mistaken for good ones? The solution is simple, indeed: as soon as the user modifies either the txtWidth or the txtHeight TextBox controls, the result fields must be cleared. In Visual Basic, you can accomplish this task by trapping each source control's *Change*

event and writing a couple of statements in the corresponding event procedure. Since *Change* is the default event for TextBox controls—just as the *Click* event is for CommandButtons controls—you only have to double-click the txtWidth and txtHeight controls on the form designer to have Visual Basic create the template for the corresponding event procedures. This is the code that you have to add to the procedures:

```
Private Sub txtWidth_Change()
    txtPerimeter.Text = ""
    txtArea.Text = ""
End Sub

Private Sub txtHeight_Change()
    txtPerimeter.Text = ""
    txtArea.Text = ""
End Sub
```

Note that you don't have to retype the statements in the *txtHeight*'s *Change* event procedure: just double-click the control to create the *Sub ... End Sub* template, and then copy and paste the code from the *txtWidth_Click* procedure. When you're finished, press F5 to run the program to check that it now behaves as expected.

The purpose of the next refinement that I am proposing is to increase the program's robustness. To see what I mean, run the Rectangle project and press the Evaluate button without entering width or height values: the program raises a Type Mismatch error when trying to extract a numeric value from the txtWidth control. If this were a real-world, compiled application, such an *untrapped* error would cause the application to end abruptly, which is, of course, unacceptable. All errors should be trapped and dealt with in a convenient way. For example, you should show the user where the problem is and how to fix it. The easiest way to achieve this is by setting up an error handler in the *cmdEvaluate_Click* procedure, as follows. (The lines you would add are in boldface.)

```
Private Sub cmdEvaluate_Click()
    ' Declare two floating point variables.
    Dim reWidth As Double, reHeight As Double
    On Error GoTo WrongValues

    ' Extract values from input textbox controls.
    reWidth = CDbl(txtWidth.Text)
    reHeight = CDbl(txtHeight.Text)
    Ensure that they are positive values.
    If reWidth <= 0 Or reHeight <= 0 Then GoTo WrongValues
    ' Evaluate results and assign to output text boxes.
    txtPerimeter.Text = CStr((reWidth + reHeight) * 2)
    txtArea.Text = CStr(reWidth * reHeight)
    Exit Sub
WrongValues:
    MsgBox "Please enter valid Width and Height values", vbExclamation
End Sub
```

Note that we have to add an *Exit Sub* statement to prevent the *MsgBox* statement from being erroneously executed during the normal execution flow. To see how the *On Error* statement works, set a breakpoint on the first line of this procedure, run the application, and press the F8 key to see what happens when either of the TextBox controls contains an empty or invalid string.

Ready, Compile, Run!

Visual Basic is a very productive programming language because it allows you to build and test your applications in a controlled environment, without first producing a compiled executable program. This is possible because Visual Basic converts your source code into *p-code* and then interprets it. P-code is a sort of intermediate language, which, because it's not executed directly by the CPU, is slower than real natively compiled code. On the other hand, the conversion from source code to p-code takes only a fraction of the time needed to deliver a compiled application. This

is a great productivity bonus unknown to many other languages. Another benefit of p-code is that you can execute it step-by-step while the program is running in the environment, investigate the values of the variables, and—to some extent—even modify the code itself. This is a capability that many other languages don't have or have acquired only recently; for example, the latest version of Microsoft Visual C++ has it. By comparison, Visual Basic has always offered this feature, which undoubtedly contributed to making it a successful language.

At some time during the program development, you might want to create an executable (EXE) program. There are several reasons to do this: compiled programs are often (much) faster than interpreted ones, users don't need to install Visual Basic to run your application, and you usually don't want to let other people peek at your source code. Visual Basic makes the compilation process a breeze: when you're sure that your application is completed, you just have to run the Make *projectname* command from the File menu.

It takes a few seconds to create the Rectangle.exe file. This executable file is independent of the Visual Basic environment and can be executed in the same way as any other Windows application—for example, from the Run command of the Start menu. But this doesn't mean that you can pass this EXE file to another user and expect that it works. All Visual Basic programs, in fact, depend on a number of ancillary files—most notably the MSVBVM60.DLL file, a part of the Visual Basic runtime—and won't execute accurately unless all such files are correctly installed on the target system.

For this reason, you should never assume that a Visual Basic program will execute on every Windows system because it's working on your computer or on other computers in your office. (If your business is software development, it's highly probable that the Visual Basic environment is installed on all the computers around you.) Instead, prepare a standard installation using the Package and Deployment Wizard, and try running your application on a clean system. If you develop software professionally, you should always have such a clean system at hand, if possible with just the operating system installed. If you're an independent developer, you probably won't be inclined to buy a complete system just to test your software. I found a very simple and relatively inexpensive solution to this dilemma: I use one computer with removable hard disks, so I can easily test my applications under different system configurations. And since a clean system requires only hundreds of megabytes of disk space, I can recycle all of my old hard disks that aren't large enough for any other use.

Before I conclude this chapter, you should be aware of one more detail. The compilation process doesn't necessarily mean that you aren't using p-code. In the Visual Basic jargon, *compiling* merely means *creating an executable file*. In fact, you can compile to p-code, even if this sounds like an oxymoron to a developer coming from another language. (See Figure 1-11.) In this case, Visual Basic creates an EXE file that embeds the same p-code that was used inside the development environment. That's why you can often hear Visual Basic developers talking about *p-code* and *native-code* compilations to better specify which type of compilation they're referring to.

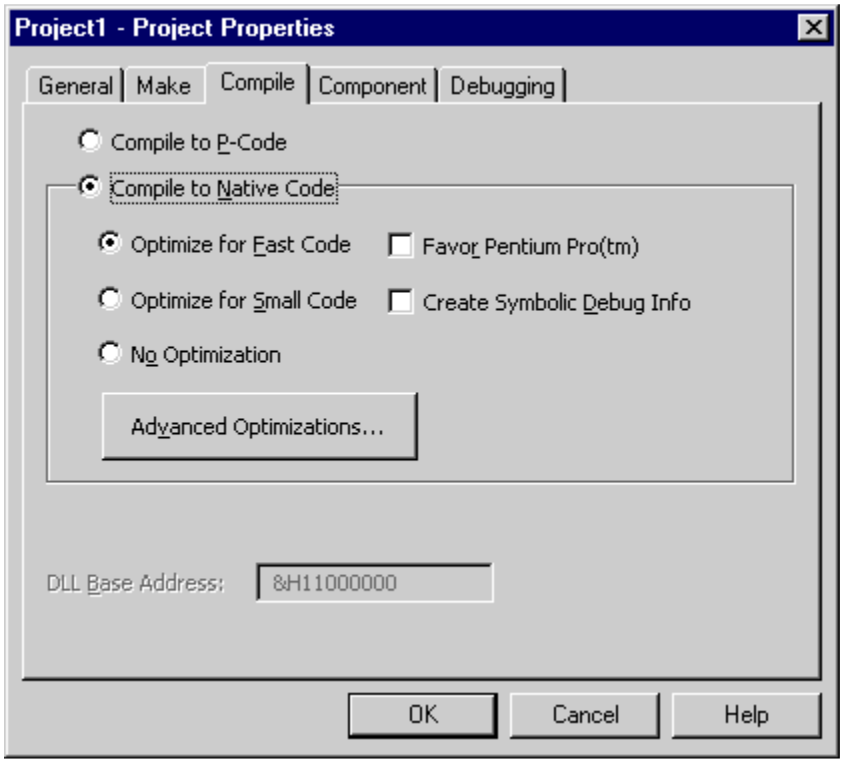

Figure 1-11. *You can opt to compile to p-code or native code in the Compile tab of the Project Properties dialog.*

In general, such p-code-compiled programs run at the same speed as interpreted programs within the IDE, so you're missing one of the biggest benefits of the compilation process. But here are a few reasons why you might decide to create a p-code executable:

- P-code-compiled executables are often smaller than programs compiled to native code. This point can be important if you're going to distribute your application over the Internet or when you're creating ActiveX controls that are embedded in an HTML page.
- P-code compilation is often faster than native code compilation, so you might prefer to stick to p-code when you compile the program in the test phase. (A few types of applications can't be tested within the IDE, most notably multithreaded components.)
- If your application spends most of its time accessing databases or redrawing windows, compilation to native code doesn't significantly improve its performance because the time spent executing Visual Basic code is only a fraction of the total execution time.

We've come to the end of this *tour de force* in the Visual Basic IDE. In this chapter, I've illustrated the basics of Visual Basic development, and I hope I've given you a taste of how productive this language can be. Now you're ready to move to the next chapters, where you can learn more about forms and controls and about how to make the best of their properties, methods, and events.