



# **Welkin Systems Limited**

天行系統有限公司

## Microsoft Excel 2016 VBA Macro Programming

如對本課程有任何意見或投訴，請電郵至  
[supervisory@welkin.com.hk](mailto:supervisory@welkin.com.hk)

Should you have any comment or complaint on  
our training courses, please email to  
[supervisory@welkin.com.hk](mailto:supervisory@welkin.com.hk)

# Table of Contents

<b>1. Introducing Visual Basic for Applications.....</b>	<b>1</b>
1.1. Getting Some BASIC Background.....	1
1.2. Delving in to VBA.....	1
1.3. Covering the Basics of VBA.....	2
1.4. Introducing the Visual Basic Editor.....	4
1.5. Working with the Project Explorer.....	6
1.6. Working with Code Windows.....	8
1.7. Customizing the VBE Environment.....	14
1.8. The Macro Recorder.....	19
1.9. About Objects and Collections.....	24
1.10. Properties and Methods.....	26
1.11. The Comment Object: A Case Study.....	27
1.12. Some Useful Application Properties.....	32
1.13. Working with Range Objects.....	33
1.14. Things to Know about Objects.....	37
<b>2. VBA Programming Fundamentals.....</b>	<b>40</b>
2.1. VBA Language Elements: An Overview.....	40
2.2. Comments.....	42
2.3. Variables, Data Types, and Constants.....	43
2.4. Assignment Statements.....	54
2.5. Arrays.....	56
2.6. Object Variables.....	57
2.7. User-Defined Data Types.....	58
2.8. Built-in Functions.....	59
2.9. Manipulating Objects and Collections.....	61
2.10. Controlling Code Execution.....	64
<b>3. Working with VBA Sub Procedures.....</b>	<b>80</b>
3.1. About Procedures.....	80
3.2. Executing Sub Procedures.....	82
3.3. Passing Arguments to Procedures.....	90
3.4. Error-Handling Techniques.....	93
3.5. A Realistic Example That Uses Sub Procedures.....	97
<b>4. Creating Function Procedures.....</b>	<b>111</b>
4.1. Sub Procedures versus Function Procedures.....	111
4.2. Why Create Custom Functions?.....	111
4.3. An Introductory Function Example.....	112
4.4. Function Procedures.....	115
4.5. Function Arguments.....	119
<b>5. VBA Programming Examples and Techniques.....</b>	<b>121</b>
5.1. Working with Ranges.....	121
5.2. Working with Workbooks and Sheets.....	143
5.3. VBA Techniques.....	146
5.4. Some Useful Functions for Use in Your Code.....	153
5.5. Some Useful Worksheet Functions.....	157
5.6. Windows API Calls.....	171
<b>6. Custom Dialog Box Alternatives.....</b>	<b>177</b>
6.1. Before You Create That UserForm.....	177
6.2. Using an Input Box.....	177
6.3. The VBA MsgBox Function.....	181
<b>7. Introducing UserForms.....</b>	<b>185</b>

# Table of Contents

7.1.	How Excel Handles Custom Dialog Boxes .....	185
7.2.	Inserting a New UserForm .....	185
7.3.	Adding Controls to a UserForm .....	186
7.4.	Toolbox Controls .....	186
7.5.	How Excel Handles Custom Dialog Box Adjusting UserForm Controls .....	190
7.6.	Adjusting a Control's Properties .....	190
7.7.	Displaying a UserForm .....	195
7.8.	Closing a UserForm .....	196
7.9.	Creating a UserForm: An Example .....	197
7.10.	Understanding UserForm Events .....	202
7.11.	Referencing UserForm Controls .....	207
7.12.	A UserForm Checklist .....	208
<b>8.</b>	<b>Working with Pivot Tables .....</b>	<b>210</b>
8.1.	An Introductory Pivot Table Example .....	210
8.2.	Creating a More Complex Pivot Table .....	215
8.3.	Creating Multiple Pivot Tables .....	219
8.4.	Creating a Reverse Pivot Table .....	222
<b>9.</b>	<b>Appendix .....</b>	<b>225</b>
9.1.	Microsoft Technical Support .....	225
9.2.	VBA Statements and Functions Reference .....	225
9.3.	Invoking Excel functions in VBA instructions .....	229
9.4.	VBA Error Codes .....	233

# 1. Introducing Visual Basic for Applications

## 1.1. Getting Some BASIC Background

Many hard-core programmers scoff at the idea of programming in BASIC. The name itself (an acronym for Beginner's All-purpose Symbolic Instruction Code) suggests that BASIC isn't a professional language. In fact, BASIC was first developed in the early 1960s as a way to teach programming techniques to college students. BASIC caught on quickly and is available in hundreds of dialects for many types of computers.

BASIC has evolved and improved over the years. For example, in many early implementations, BASIC was an interpreted language. Each line was interpreted before it was executed, causing slow performance. Most modern dialects of BASIC allow the code to be compiled — converted to machine code — which results in faster and more efficient execution.

BASIC gained quite a bit of respectability in 1991 when Microsoft released Visual Basic for Windows. This product made it easy for the masses to develop stand-alone applications for Windows. Visual Basic has very little in common with early versions of BASIC, but Visual Basic is the foundation on which VBA was built.

## 1.2. Delving in to VBA

Excel 5 was the first application on the market to feature Visual Basic for Applications (VBA). VBA is best thought of as Microsoft's common application scripting language, and it's included with most Office 2010 applications and even in applications from other vendors. Therefore, if you master VBA by using Excel, you'll be able to jump right in and write macros for other Microsoft (and some non-Microsoft) products. Even better, you'll be able to create complete solutions that use features across various applications.

### Object models

The secret to using VBA with other applications lies in understanding the object model for each application. VBA, after all, simply manipulates objects, and each product (Excel, Word, Access, PowerPoint, and so on) has its own unique object model. You can program an application by using the objects that the application exposes.

Excel's object model, for example, exposes several very powerful data analysis objects, such as worksheets, charts, pivot tables, and numerous mathematical, financial, engineering, and general business functions. With VBA, you can work with these objects and develop automated procedures. While you work with VBA in Excel, you gradually build an understanding of the object model. Warning: The object model will be very confusing at first. Eventually, however, the pieces come together — and all of a sudden, you realize that you've mastered it!

### VBA versus XLM

Before version 5, Excel used a powerful (but very cryptic) macro language called XLM. Later versions of Excel (including Excel 2010) still execute XLM macros, but the capability to record macros in XLM was removed beginning with Excel 97. As a developer, you should be aware of XLM (in case you ever encounter

### 1.3. Covering the Basics of VBA

macros written in that system), but you should use VBA for your development work.

Before I get into the meat of things, I suggest that you read through the material in this section to get a broad overview of where I'm heading. I cover these topics in the remainder of this chapter.

Following is a quick-and-dirty summary of what VBA is all about:

- **Code:** You perform actions in VBA by executing VBA code. You write (or record) VBA code, which is stored in a VBA module.
- **Module:** VBA modules are stored in an Excel workbook file, but you view or edit a module by using the Visual Basic Editor (VBE). A VBA module consists of procedures.
- **Procedures:** A procedure is basically a unit of computer code that performs some action. VBA supports two types of procedures: Sub procedures and Function procedures.
- **Sub:** A Sub procedure consists of a series of statements and can be executed in a number of ways. Here's an example of a simple Sub procedure called Test: This procedure calculates a simple sum and then displays the result in a message box.

```
Sub Test()
```

```
    Sum = 1 + 1
```

```
    MsgBox "The answer is " & Sum
```

```
End Sub
```

- **Function:** A VBA module can also have Function procedures. A Function procedure returns a single value (or possibly an array). A Function can be called from another VBA procedure or used in a worksheet formula. Here's an example of a Function named AddTwo:

```
Function AddTwo(arg1, arg2)
```

```
    AddTwo = arg1 + arg2
```

```
End Function
```

- **Objects:** VBA manipulates objects contained in its host application. (In this case, Excel is the host application.) Excel provides you with more than 100 classes of objects to manipulate. Examples of objects include a workbook, a worksheet, a range on a worksheet, a chart, and a shape. Many more objects are at your disposal, and you can use VBA code to manipulate them. Object classes are arranged in a hierarchy.

Objects also can act as containers for other objects. For example, Excel is an object called Application, and it contains other objects, such as Workbook objects. The Workbook object contains other objects, such as Worksheet objects and Chart objects. A Worksheet object contains objects such as Range objects, PivotTable objects, and so on. The arrangement of these objects is referred to as Excel's object model.

- **Collections:** Like objects form a collection. For example, the Worksheets collection consists of all the worksheets in a particular workbook. Collections are objects in themselves.
- **Object hierarchy:** When you refer to a contained or member object, you specify its position in the object hierarchy by using a period (also known

as a dot) as a separator between the container and the member. For example, you can refer to a workbook named Book1.xlsx as

*Application.Workbooks("Book1.xlsx")*

This code refers to the Book1.xlsx workbook in the Workbooks collection. The Workbooks collection is contained in the Excel Application object. Extending this type of referencing to another level, you can refer to Sheet1 in Book1 as

*Application.Workbooks("Book1.xlsx").Worksheets("Sheet1")*

You can take it to still another level and refer to a specific cell as follows:

*Application.Workbooks("Book1.xlsx").Worksheets("Sheet1").Range("A1")*

- **Active objects:** If you omit a specific reference to an object, Excel uses the active objects. If Book1 is the active workbook, the preceding reference can be simplified as

*Worksheets("Sheet1").Range("A1")*

If you know that Sheet1 is the active sheet, you can simplify the reference even more:

*Range("A1")*

- **Objects properties:** Objects have properties. A property can be thought of as a setting for an object. For example, a range object has properties such as Value and Address. A chart object has properties such as HasTitle and Type. You can use VBA to determine object properties and also to change them. Some properties are read-only properties and can't be changed by using VBA.

You refer to properties by combining the object with the property, separated by a period. For example, you can refer to the value in cell A1 on Sheet1 as

*Worksheets("Sheet1").Range("A1").Value*

- **VBA variables:** You can assign values to VBA variables. Think of a variable as a name that you can use to store a particular value. To assign the value in cell A1 on Sheet1 to a variable called Interest, use the following VBA statement:

*Interest = Worksheets("Sheet1").Range("A1").Value*

- **Object methods:** Objects have methods. A method is an action that is performed with the object. For example, one of the methods for a Range object is ClearContents. This method clears the contents of the range. You specify methods by combining the object with the method, separated by a period. For example, to clear the contents of cell A1 on the active worksheet, use

*Range("A1").ClearContents*

## 1.4. Introducing the Visual Basic Editor

- **Standard programming constructs:** VBA also includes many constructs found in modern programming languages, including arrays, loops, and so on.
- **Events:** Some objects recognize specific events, and you can write VBA code that is executed when the event occurs. For example, opening a workbook triggers a `Workbook_Open` event. Changing a cell in a worksheet triggers a `Worksheet_Change` event.

Believe it or not, the preceding section pretty much describes VBA. Now it's just a matter of learning the details.

All your VBA work is done in the Visual Basic Editor (VBE). The VBE is a separate application that works seamlessly with Excel. By seamlessly, I mean that Excel takes care of the details of opening the VBE when you need it. You can't run VBE separately; Excel must be running in order for the VBE to run.

**Notes: VBA modules are stored in workbook files. However, the VBA modules aren't visible unless you activate the VBE.**

### Displaying Excel's Developer tab

The Excel Ribbon doesn't display the Developer tab by default. If you're going to be working with VBA, it's essential that you turn on the Developer tab:

1. Right-click the Ribbon and choose *Customize the Ribbon*.  
Excel displays the *Customize Ribbon* tab of the *Excel Options* dialog box.
2. In the list box on the right, place a checkmark next to *Developer*.
3. Click *OK*.

After you perform these steps, Excel displays a new tab, as shown in Figure 7-1.



FIGURE 7-1: By default, the Developer tab is not displayed.

### Activating the VBE

When you're working in Excel, you can switch to the VBE by using either of the following techniques:

- Press **Alt+F11**.
- Choose **Developer⇒Code⇒Visual Basic**.

In addition, you can access two special modules as follows. (These special VBA modules are used for event-handler procedures, which I describe in Chapter 19.)

- Right-click a sheet tab and choose *View Code*, which takes you to the code module for the sheet.
- Right-click a workbook's title bar and choose *View Code*, which takes you to the code module for the workbook. If the workbook window is maximized in Excel, the workbook window's title bar is not visible.



Figure 7-2 shows the VBE. Chances are that your VBE window won't look exactly like the window shown in the figure. This window is highly customizable — you can hide windows, change their sizes, dock them, rearrange them, and so on.

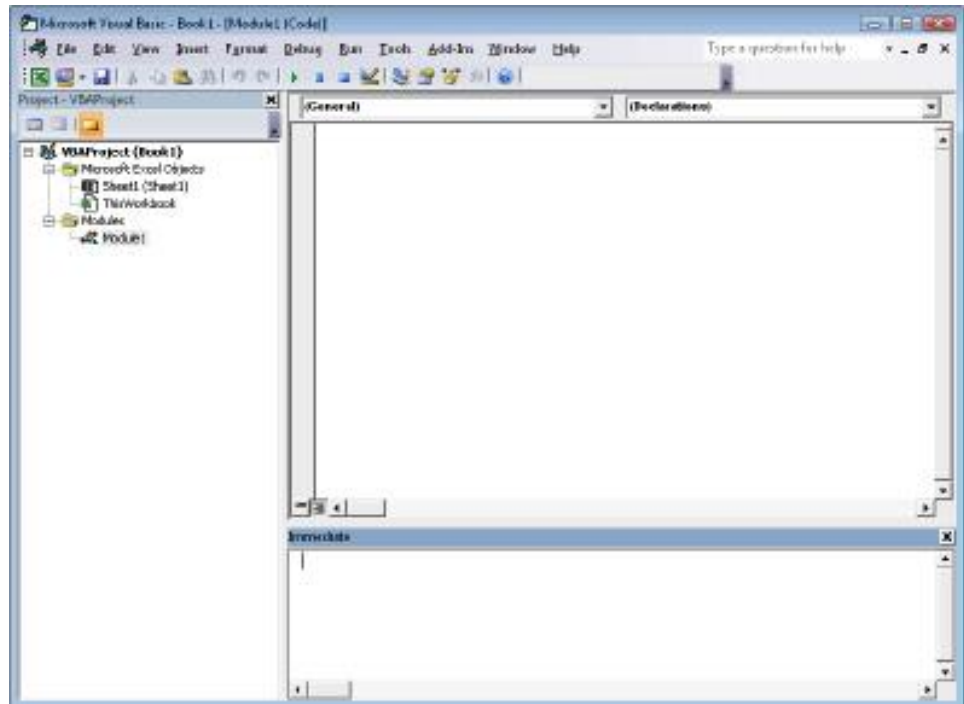


FIGURE 7-2: The Visual Basic Editor window.

### The VBE windows

The VBE has a number of parts. I briefly describe some of the key components in the following list:

- VBE menu bar: Although Excel uses a fancy new Ribbon interface, the VBE is still stuck in the menu and toolbar world. The VBE menu bar works like every other menu bar that you've encountered. It contains commands that you use to work with the various components in the VBE. Also, you'll find that many of the menu commands have shortcut keys associated with them. For example, the View⇒Immediate Window command has a shortcut key of Ctrl+G.

**Notes: The VBE also features shortcut menus. As you'll discover, you can right-click virtually anything in a VBE window to get a shortcut menu of common commands.**

- VBE toolbars: The Standard toolbar, which is directly under the menu bar by default, is one of six VBE toolbars available. (The menu bar is also considered a toolbar.) You can customize toolbars, move them around, display other toolbars, and so on. Choose View⇒Toolbars⇒Customize to work with VBE toolbars.
- Project Explorer window: The Project Explorer window displays a tree diagram that consists of every workbook that is currently open in Excel (including add-ins and hidden workbooks). Each workbook is known as a project. I discuss the Project Explorer window in more detail in the next section ("Working with the Project Explorer").

If the Project Explorer window isn't visible, press Ctrl+R. To hide the Project Explorer window, click the Close button in its title bar or right-click anywhere in the Project Explorer window and select Hide from the shortcut menu.

## 1.5. Working with the Project Explorer

- **Code window:** A Code window (sometimes known as a Module window) contains VBA code. Every item in a project's tree has an associated code window. To view a code window for an object, double-click the object in the Project Explorer window. For example, to view the code window for the Sheet1 object, double-click Sheet1 in the Project Explorer window. Unless you've added some VBA code, the Code window is empty.

Another way to view the Code window for an object is to select the object in the Project Explorer window and then click the View Code button in the toolbar at the top of the Project Explorer window.

I discuss Code windows later in this chapter (see "Working with Code Windows").

- **Immediate window:** The Immediate window is most useful for executing VBA statements directly, testing statements, and debugging your code. This window may or may not be visible. If the Immediate window isn't visible, press Ctrl+G. To close the Immediate window, click the Close button in its title bar (or right-click anywhere in the Immediate window and select Hide from the shortcut menu).

When you're working in the VBE, each Excel workbook and add-in that's currently open is considered a project. You can think of a project as a collection of objects arranged as an expandable tree. You can expand a project by clicking the plus sign (+) at the left of the project's name in the Project Explorer window. You contract a project by clicking the minus sign (–) to the left of a project's name. If you try to expand a project that's protected with a password, you're prompted to enter the password.

**Notes:** The top of the Project Explorer window contains three icons. The third icon, named **Toggle Folder**, controls whether the objects in a project are displayed in a hierarchy or are shown in a single nonhierarchical list.

Figure 7-3 shows a Project Explorer window with four projects listed (two XLAM add-ins and two workbooks).

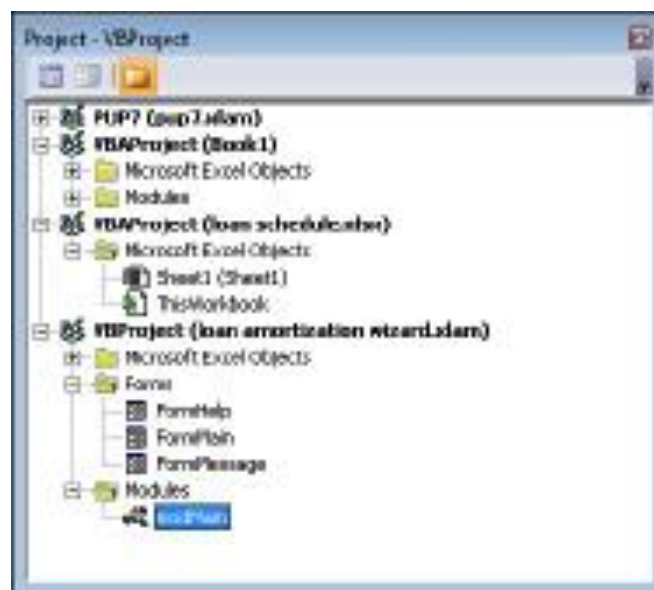


FIGURE 7-3: A Project Explorer window with four projects listed.

**Notes:** When you activate the VBE, you can't assume that the code module that's displayed corresponds to the highlighted object in the Project Explorer window. To make sure that you're working in the correct code module, always double-click the object in the Project Explorer window.

If you have many workbooks and add-ins loaded, the Project Explorer window can be a bit overwhelming. Unfortunately, you can't hide projects in the Project Explorer window. However, you probably want to keep the project outlines contracted if you're not working on them.

When viewing the Project Explorer in folder view, every project expands to show at least one node called Microsoft Excel Objects. This node expands to show an item for each worksheet and chart sheet in the workbook (each sheet is considered an object) and another object called ThisWorkbook (which represents the Workbook object). If the project has any VBA modules, the project listing also shows a Modules node, and the modules are listed there. A project can also contain a node called Forms that contains UserForm objects (also known as custom dialog boxes). If your project has any class modules, it displays another node called Class Modules. Similarly, if your project has any references, you see another node called References. The References node is a bit misleading because references can't contain any VBA code.

### **Adding a new VBA module**

To add a new VBA module to a project, select the project's name in the Project Explorer window and choose Insert⇒Module. Or you can just right-click the project's name and choose Insert⇒Module from the shortcut menu.

When you record a macro, Excel automatically inserts a VBA module to hold the recorded code.

### **Removing a VBA module**

If you need to remove a VBA module, a class module, or a UserForm from a project, select the module's name in the Project Explorer window and choose File⇒Remove xxx (where xxx is the name of the module). Or you can right-click the module's name and choose Remove xxx from the shortcut menu. You're asked whether you want to export the module before removing it. (See the next section for details.)

You can't remove code modules associated with the workbook (the ThisWorkbook code module) or with a sheet (for example, the Sheet1 code module).

### **Exporting and importing objects**

Except for those listed under the References node, you can save every object in a project to a separate file. Saving an individual object in a project is called exporting. It stands to reason that you can also import objects into a project. Exporting and importing objects might be useful if you want to use a particular object (such as a VBA module or a UserForm) in a different project.

To export an object, select it in the Project Explorer window and choose File⇒Export File. You get a dialog box that asks for a filename. Note that the object remains in the project. (Only a copy of it is exported.) If you export a UserForm object, any code associated with the UserForm is also exported.

To import a file into a project, select the project's name in the Project Explorer window and choose File⇒Import File. You get a dialog box that asks for a file. You can import only a file that has been exported by choosing the File⇒Export File command.

**Notes:** If you want to copy a module or UserForm to another project, you don't need to export and then import the object. Make sure that both projects are open; then simply activate the Project Explorer and drag the object from one project to the other. The original module or UserForm remains, and a copy is added to the other project.

## 1.6. Working with Code Windows

When you become proficient with VBA, you'll be spending lots of time working in code windows. Each object in a project has an associated code window. To summarize, these objects can be

- The workbook itself (ThisWorkbook in the Project Explorer window)
- A worksheet or chart sheet in a workbook (for example, Sheet1 or Chart1 in the Project Explorer window)
- A VBA module
- A class module (a special type of module that lets you create new object classes)
- A UserForm

### Minimizing and maximizing windows

Depending on how many workbooks and add-ins are open, the VBE can have lots of code windows, and things can get a bit confusing. Code windows are much like worksheet windows in Excel. You can minimize them, maximize them, rearrange them, and so on. Most people find it most efficient to maximize the Code window that they're working in. Doing so enables you to see more code and keeps you from getting distracted. To maximize a Code window, click the maximize button in its title bar or just double-click its title bar. To restore a Code window (make it nonmaximized), click the Restore button (below the Application title bar).

Sometimes, you may want to have two or more Code windows visible. For example, you might want to compare the code in two modules or perhaps copy code from one module to another. To view two or more Code windows at once, make sure that the active code window isn't maximized. Then drag and resize the windows that you want to view.

Minimizing a code window gets it out of the way. You can also click the Close button in a Code window's title bar to close the window completely. To open it again, just double-click the appropriate object in the Project Explorer window.

The VBE doesn't have a menu command to close a workbook. You must reactivate Excel and close it from there. You can, however, use the Immediate window to close a workbook or an add-in. Just activate the Immediate window (press Ctrl+G if it's not visible), type a VBA statement like the one that follows, and press Enter:

```
Workbooks("myaddin.xlam").Close
```

As you'll see, this statement executes the Close method of the Workbook object, which closes a workbook. In this case, the workbook happens to be an add-in.

### Storing VBA code

In general, a code window can hold four types of code:

- **Sub procedures:** A procedure is a set of instructions that performs some action.
- **Function procedures:** A function is a set of instructions that returns a single value or an array (similar in concept to a worksheet function, such as SUM).
- **Property procedures:** These are special procedures used in class modules.

- **Declarations:** A declaration is information about a variable that you provide to VBA. For example, you can declare the data type for variables you plan to use.

A single VBA module can store any number of Sub procedures, Function procedures, and declarations. How you organize a VBA module is completely up to you. Some people prefer to keep all their VBA code for an application in a single VBA module; others like to split up the code into several different modules.

### Entering VBA code

Before you can do anything meaningful, you must have some VBA code in a Code window. This VBA code must be within a procedure. A procedure consists of VBA statements. For now, I focus on one type of Code window: a VBA module.

You can add code to a VBA module in three ways:

- Enter the code manually. Use your keyboard to type your code.
- Use the macro-recorder feature. Use Excel's macro-recorder feature to record your actions and convert them into VBA code.
- Copy and paste. Copy the code from another module and paste it into the module that you're working in.

### Entering code manually

Sometimes, the most direct route is the best one. Entering code directly involves . . . well, entering the code directly. In other words, you type the code by using your keyboard. You can use the Tab key to indent the lines that logically belong together — for example, the conditional statements between the If and End If statements. Indenting isn't necessary, but it makes the code easier to read, so it's a good habit to acquire.

Entering and editing text in a VBA module works just as you would expect. You can select text, copy it or cut it, and then paste it to another location.

A single instruction in VBA can be as long as you need it to be. For readability's sake, however, you may want to break a lengthy instruction into two or more lines. To do so, end the line with a space followed by an underscore character and then press Enter and continue the instruction on the following line. The following code, for example, is a single VBA statement split over four lines:

```
MsgBox "Can't find " & UCase(SHORTCUTMENUFILE) _  
& vbCrLf & vbCrLf & "The file should be located in " _  
& ThisWorkbook.Path & vbCrLf & vbCrLf _  
& "You may need to reinstall BudgetMan", vbCritical, APPNAME
```

Notice that I indented the last three lines of this statement. Doing so is optional, but it helps clarify the fact that these four lines are, in fact, a single statement.

To get a feel for entering a VBA procedure, try this: Insert a VBA module into a project and then enter the following procedure into the Code window of the module:

```
Sub SayHello()  
Msg = "Is your name " & Application.UserName & "?"  
Ans = MsgBox(Msg, vbYesNo)  
If Ans = vbNo Then  
MsgBox "Oh, never mind."  
Else
```

```
MsgBox "I must be clairvoyant!"  
End If  
End Sub
```

Figure 7-4 shows how this code looks in a VBA module.

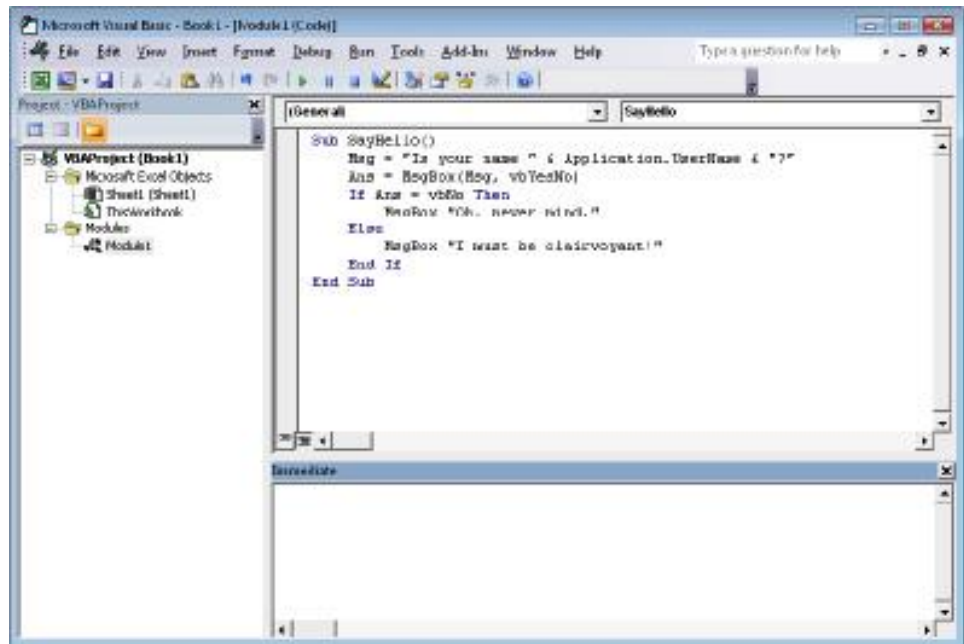


FIGURE 7-4: Your first VBA procedure.

To execute the SayHello procedure, make sure that the cursor is located anywhere within the text that you typed. Then do any of the following:

- Press F5.
- Choose Run⇒Run Sub/UserForm.
- Click the Run Sub/UserForm button on the Standard toolbar.

If you entered the code correctly, the procedure executes, and you can respond to a simple dialog box (see Figure 7-5) that displays the username, as listed in the Excel Options dialog box. Notice that Excel is activated when the macro executes. At this point, it's not important that you understand how the code works; that becomes clear later in this chapter and in subsequent chapters.



FIGURE 7-5: The result of running the procedure in Figure 7-4.

What you did in this exercise was write a VBA Sub procedure (also known as a macro). When you issued the command to execute the macro, the VBE quickly compiled the code and executed it. In other words, each instruction was evaluated, and Excel simply did what it was told to do. You can execute this macro any number of times, although it tends to lose its appeal after a while.

For the record, this simple procedure uses the following concepts (all of which I cover later in the book):

- Declaring a procedure (the first line)
- Assigning a value to variables (Msg and Ans)
- Concatenating strings (using the & operator)
- Using a built-in VBA function (MsgBox)
- Using built-in VBA constants (vbYesNo and vbNo)
- Using an If-Then-Else construct
- Ending a procedure (the last line)

### Using the macro recorder

Another way to get code into a VBA module is to record your actions by using the Excel macro recorder.

No matter how hard you try, there is absolutely no way to record the SayHello procedure shown in the previous section. As you'll see, recording macros is very useful, but it has some limitations. In fact, when you record a macro, you almost always need to make adjustments or enter some code manually.

This next example shows how to record a macro that simply changes the page setup to landscape orientation. If you want to try these, start with a blank workbook:

1. Activate a worksheet in the workbook (any worksheet will do).
2. Choose Developer⇒Code⇒Record Macro.

Excel displays its Record Macro dialog box.

3. Click OK to accept the default setting for the macro.

Excel automatically inserts a new VBA module into the workbook's VBA project. From this point on, Excel converts your actions into VBA code. Notice that Excel's status bar displays a blue square. You can click that control to stop recording.

4. Choose Page Layout⇒Page Setup⇒Orientation⇒Landscape.
5. Select Developer⇒Code⇒Stop Recording (or click the blue square in the status bar).

Excel stops recording your actions.

To view the macro, activate the VBE (pressing Alt+F11 is the easiest way) and locate the project in the Project Explorer window. Double-click the Modules node to expand it. Then double-click the Module1 item to display the code window. (If the project already had a Module1, the new macro will be in Module2.) The code generated by this single Excel command is shown in Figure 7-6. Remember that code lines preceded by an apostrophe are comments and are not executed.

You may be surprised by the amount of code generated by this single command. (I know I was the first time I tried something like this.) Although you changed only one simple setting in the Page Setup tab, Excel generates more than 50 lines of code that affects dozens of print settings.

This code listing brings up an important concept. The Excel macro recorder is not the most efficient way to generate VBA code. More often than not, the code produced when you record a macro is overkill. Consider the recorded macro that switches to landscape mode. Practically every statement in that macro is extraneous. You can simplify this macro considerably by deleting the extraneous code. Deleting extraneous code makes the macro easier to read, and the macro also runs a bit faster because it doesn't do things that are unnecessary. In fact, you can simplify this recorded macro to the following:

```
Sub Macro1()  
    With ActiveSheet.PageSetup  
        .Orientation = xlLandscape  
    End With  
End Sub
```

I deleted all the code except for the line that sets the Orientation property. Actually, you can simplify this macro even more because the With-End With construct isn't necessary when you're changing only one property:

```
Sub Macro1()  
    ActiveSheet.PageSetup.Orientation = xlLandscape  
End Sub
```

In this example, the macro changes the Orientation property of the PageSetup object on the active sheet. By the way, xlLandscape is a built-in constant that's provided to make things easier for you. The variable xlLandscape has a value of 2, and xlPortrait has a value of 1. The following macro works the same as the preceding Macro1:

```
Sub Macro1a()  
    ActiveSheet.PageSetup.Orientation = 2  
End Sub
```





FIGURE 7-6: Code generated by Excel's macro recorder.

Most would agree that it's easier to remember the name of the constant than the arbitrary numbers. You can use the Help system to learn the relevant constants for a particular command.

You could have entered this procedure directly into a VBA module. To do so, you would have to know which objects, properties, and methods to use. Obviously, recording the macro is much faster, and this example has a built-in bonus: You also learned that the PageSetup object has an Orientation property.

### Copying VBA code

So far, I've covered typing code directly into a module and recording your actions to generate VBA code. The final method of getting code into a VBA module is to copy it from another module. For example, you may have written a procedure for one project that would also be useful in your current project. Rather than re-enter the code, you can simply open the workbook, activate the module, and use the

## 1.7. Customizing the VBE Environment

normal Clipboard copy-and-paste procedures to copy it into your current VBA module. After you've finished pasting, you can modify the code as necessary.

And don't forget about the Internet. You'll find thousands of VBA code examples at Web sites, forums, and blogs. It's a simple matter to copy code from a browser and paste it into a VBA module.

If you're serious about becoming an Excel programmer, you'll be spending a lot of time with the VBE window. To help make things as comfortable as possible, the VBE provides quite a few customization options.

When the VBE is active, choose Tools⇒Options. You see a dialog box with four tabs: Editor, Editor Format, General, and Docking. I discuss some of the most useful options on these tabs in the sections that follow. By the way, don't confuse this Options dialog box with the Excel Options dialog box, which you bring up by choosing Office⇒Excel Options in Excel.

### Using the Editor tab

Figure 7-7 shows the options that you access by clicking the Editor tab of the Options dialog box.



FIGURE 7-7: The Editor tab of the Options dialog box.

### Auto Syntax Check option

The Auto Syntax Check setting determines whether the VBE pops up a dialog box if it discovers a syntax error while you're entering your VBA code. The dialog box tells you roughly what the problem is. If you don't choose this setting, VBE flags syntax errors by displaying them in a different color from the rest of the code, and you don't have to deal with any dialog boxes popping up on your screen.

I keep this setting turned off because I find the dialog boxes annoying, and I can usually figure out what's wrong with an instruction. But if you're new to VBA, you might find the Auto Syntax Check assistance helpful.

### Require Variable Declaration option

If the Require Variable Declaration option is set, VBE inserts the following statement at the beginning of each new VBA module that you insert:

*Option Explicit*

If this statement appears in your module, you must explicitly define each variable that you use. Variable declaration is an excellent habit to get into, although it does require additional effort on your part. If you don't declare your variables, they will all be of the Variant data type, which is flexible but not efficient in terms of storage or speed. I discuss variable declaration in more depth in Chapter 8.

**Notes: Changing the Require Variable Declaration option affects only new modules, not existing modules.**

### Auto List Members option

If the Auto List Members option is set, VBE provides help when you're entering your VBA code by displaying a list of member items for an object. These items include methods and properties for the object that you typed.

This option is very helpful, and I always keep it turned on. Figure 7-8 shows an example of Auto List Members (which will make a lot more sense when you actually start writing VBA code). In this example, VBE is displaying a list of members for the Application object. The list changes as you type additional characters, showing only the members that begin with the characters you type. You can just select an item from the list and press Tab (or double-click the item), thus avoiding typing it. Using the Auto List Members list also ensures that the item is spelled correctly.



FIGURE 7-8: An example of Auto List Members.

### Auto Quick Info option

If the Auto Quick Info option is set, the VBE displays information about the arguments available for functions, properties, and methods while you type. This information can be very helpful, and I always leave this setting on. Figure 7-9 shows this feature in action. It's displaying the syntax for the Cells property.

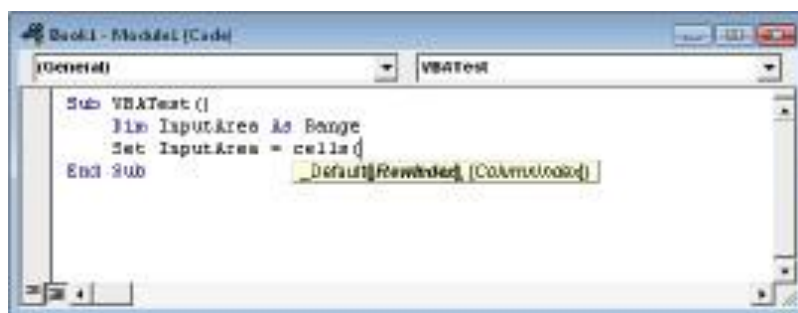


FIGURE 7-9: An example of Auto Quick Info offering help about the Cells property.

**Auto Data Tips option**

If the Auto Data Tips option is set, you can hover your mouse pointer over a variable, and VBE displays the value of the variable. This technique works only when the procedure is paused while debugging. When you enter the wonderful world of debugging, you'll definitely appreciate this option. I always keep this option turned on.

**Auto Indent option**

The Auto Indent setting determines whether VBE automatically indents each new line of code by the same amount as the previous line. I'm a big fan of using indentations in my code, so I keep this option on. You can also specify the number of characters to indent; the default is four.

**Notes: Use the Tab key, not the space bar, to indent your code. Using the Tab key results in more consistent spacing. In addition, you can use Shift+Tab to unindent a line of code. These keys also work if you select more than one statement.**

**Drag-and-Drop Text Editing option**

The Drag-and-Drop Text Editing option, when enabled, lets you copy and move text by dragging and dropping. I keep this option turned on, but I never use drag-and-drop editing. I prefer to use keyboard shortcuts for copying and pasting.

**Default to Full Module View option**

The Default to Full Module View option specifies how procedures are viewed. If this option is set, procedures in the code window appear as a single scrollable window. If this option is turned off, you can see only one procedure at a time. I keep this setting turned on.

**Procedure Separator option**

When the Procedure Separator option is turned on, the VBE displays separator bars between procedures in a code window (assuming that the Default to Full Module View option is also selected). I like the visual cues that show where my procedures end, so I keep this option turned on.

**Using the Editor Format tab**

Figure 7-10 shows the Editor Format tab of the Options dialog box. The options on this tab control the appearance of the VBE itself.

- **Code Colors option:** The Code Colors option lets you set the text color (foreground and background) and the indicator color displayed for various elements of VBA code. Choosing these colors is largely a matter of individual preference. Personally, I find the default colors to be just fine. But for a change of scenery, I occasionally play around with these settings.
- **Font option:** The Font option lets you select the font that's used in your VBA modules. For best results, stick with a fixed-width font (monofont) such as Courier New. In a fixed-width font, all characters are exactly the same width. Using fixed-width characters makes your code much more readable because the characters are nicely aligned vertically and you can easily distinguish multiple spaces.
- **Size setting:** The Size setting specifies the size of the font in the VBA modules. This setting is a matter of personal preference determined by

your video display resolution and your eyesight. The default size of 10 (points) works for me.

- **Margin Indicator Bar option:** The Margin Indicator Bar option controls the display of the vertical margin indicator bar in your modules. You should keep this turned on; otherwise, you won't be able to see the helpful graphical indicators when you're debugging your code.



FIGURE 7-10: The Editor Format tab of the Options dialog box.

### Using the General tab

Figure 7-11 shows the following options available under the General tab in the Options dialog box:

- **Form Grid Settings:** The options in this section are for UserForms (custom dialog boxes); they let you specify a grid to help align controls on the UserForm. When you have some experience designing UserForms, you can determine whether a grid display is helpful or not.
- **Show ToolTips check box:** This checkbox refers to toolbar buttons. There's no reason to turn off the ToolTips display.
- **Collapse Proj. Hides Windows option:** If checked, this setting causes the windows to close automatically when you collapse a project in the Project window. I keep this setting turned on.
- **Edit and Continue section:** This area contains one option, which may be useful for debugging. When checked, VBA displays a message if your variables are about to lose their values because of a problem.
- **Error Trapping settings:** These settings determine what happens when an error is encountered. If you write any error-handling code, make sure that the Break on Unhandled Errors option is set. If the Break on All Errors option is set, error-handling code is ignored (which is hardly ever what you want). I discuss error-handling techniques in Chapter 9.
- **Compile settings:** The two Compile settings deal with compiling your code. I keep both of these options turned on. Compiling code is virtually instantaneous unless the project is extremely large.

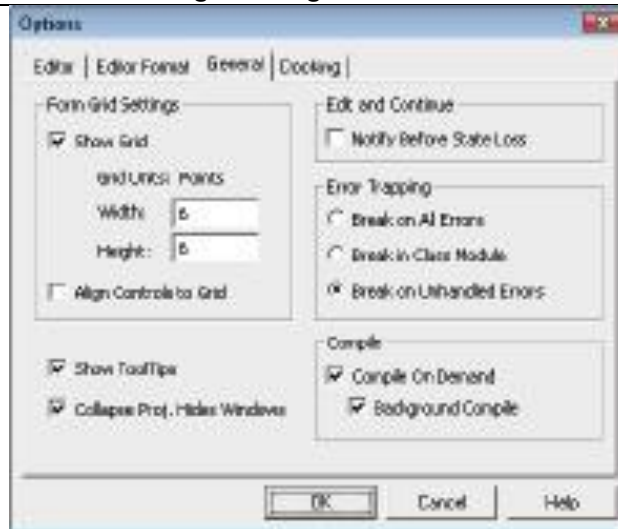


FIGURE 7-11: The General tab of the Options dialog box.

### Using the Docking tab

Figure 7-12 shows the Docking tab of the Options dialog box. These options determine how the various windows in the VBE behave. When a window is docked, it's fixed in place along one of the edges of the VBE window. Docking windows makes it much easier to identify and locate a particular window. If you turn off all docking, you have a big mess of windows that are very confusing. Generally, you'll find that the default settings work fine.

To dock a window, just drag it to the desired location. For example, you might want to dock the Project Explorer window to the left side of the screen. Just drag its title bar to the left, and you see an outline that shows it docked. Release the mouse, and the window is docked.

**Notes:** Docking windows in the VBE has always been a bit problematic. Often, you find that some windows simply refuse to be docked. I've found that if you persist long enough, the procedure will eventually work. Unfortunately, I don't have any secret window-docking techniques.

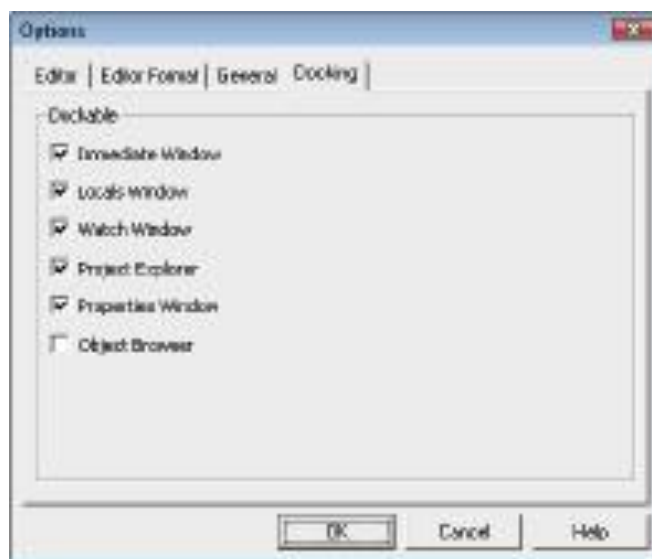


FIGURE 7-12: The Docking tab of the Options dialog box.

## 1.8. The Macro Recorder

Earlier in this chapter, I discuss the macro recorder, which is a tool that converts your Excel actions into VBA code. This section covers the macro recorder in more detail.

**Notes: This is another reminder to make sure that Excel displays the Developer tab in the Ribbon. If you don't see this tab, refer to "Displaying Excel's Developer tab" earlier in this chapter.**

The macro recorder is an extremely useful tool, but remember the following points:

- The macro recorder is appropriate only for simple macros or for recording a small part of a more complex macro.
- Not all the actions you make in Excel get recorded.
- The macro recorder can't generate code that performs looping (that is, repeating statements), assigns variables, executes statements conditionally, displays dialog boxes, and so on.
- The macro recorder always creates Sub procedures. You can't create a Function procedure by using the macro recorder.
- The code that is generated depends on certain settings that you specify.
- You'll often want to clean up the recorded code to remove extraneous commands.

### What the macro recorder actually records

The Excel macro recorder translates your mouse and keyboard actions into VBA code. I could probably write several pages describing how this translation occurs, but the best way to show you is by example. Follow these steps:

1. Start with a blank workbook.
2. Make sure that the Excel window isn't maximized.  
You don't want it to fill the entire screen.
3. Press Alt+F11 to activate the VBE window.
4. Resize and arrange Excel's window and the VBE window so that both are visible. (For best results, minimize any other applications that are running.)
5. Activate Excel, choose Developer⇒Code⇒Record Macro and then click OK to start the macro recorder.
6. Activate the VBE window.
7. In the Project Explorer window, double-click Module1 to display that module in the code window.
8. Close the Project Explorer window in the VBE to maximize the view of the code window.

Your screen layout should look something like the example in Figure 7-13. The size of the windows depends on your video resolution. If you happen to have a dual display system, just put the VBA window on one display and the Excel window on the other display.

Now move around in the worksheet and select various Excel commands. Watch while the code is generated in the window that displays the VBA module. Select cells, enter data, format cells, use the Ribbon commands, create a chart, manipulate graphic objects, and so on. I guarantee that you'll be enlightened while you watch the code being spit out before your very eyes.



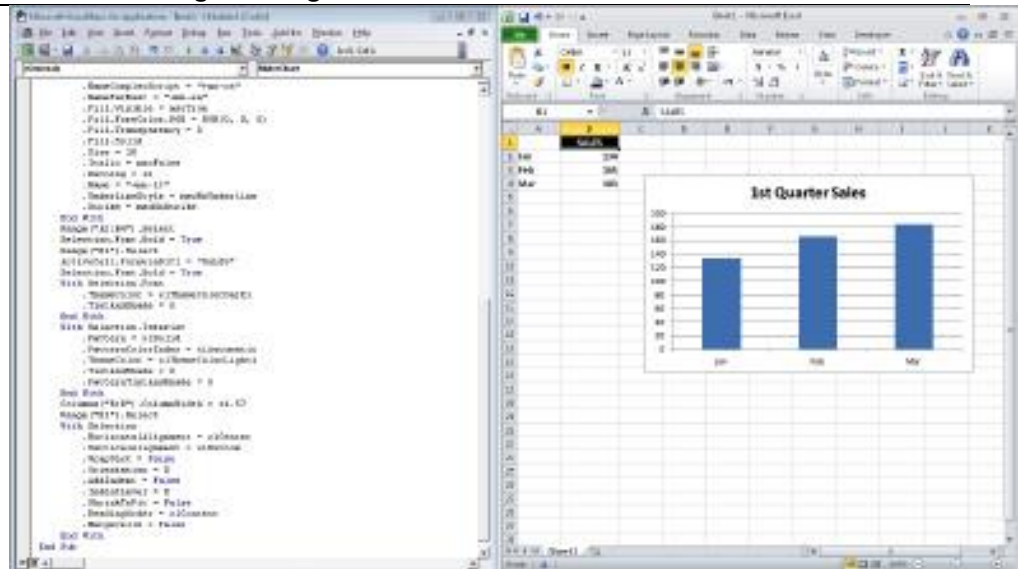


FIGURE 7-13: A convenient window arrangement for watching the macro recorder do its thing.

### Relative or absolute recording?

When recording your actions, Excel normally records absolute references to cells. In other words, when you select a cell, it will remember that exact cell (not the cell relative to the current active cell). To demonstrate how absolute references work, perform these steps and examine the code:

1. Activate a worksheet and start the macro recorder.
2. Activate cell B1.
3. Enter Jan into cell B1.
4. Move to cell C1 and enter Feb.
5. Continue this process until you've entered the first six months of the year in B1:G1.
6. Click cell B1 to activate it again.
7. Stop the macro recorder and examine the new code in the VBE.

Excel generates the following code:

```

Sub Macro1()
    Range("B1").Select
    ActiveCell.FormulaR1C1 = "Jan"
    Range("C1").Select
    ActiveCell.FormulaR1C1 = "Feb"
    Range("D1").Select
    ActiveCell.FormulaR1C1 = "Mar"
    Range("E1").Select
    ActiveCell.FormulaR1C1 = "Apr"
    Range("F1").Select
    ActiveCell.FormulaR1C1 = "May"

```



```
Range("G1").Select
ActiveCell.FormulaR1C1 = "Jun"
Range("B1").Select
End Sub
```

To execute this macro from within Excel, choose Developer⇒Code⇒Macros (or press Alt+F8) and select Macro1 (or whatever the macro is named) and click the Run button.

The macro, when executed, re-creates the actions that you performed when you recorded it. These same actions occur regardless of which cell is active when you execute the macro. Recording a macro using absolute references always produces the exact same results.

In some cases, however, you want your recorded macro to work with cell locations in a relative manner. For example, you'd probably want such a macro to start entering the month names in the active cell. In such a case, you want to use relative recording to record the macro.

You control how references are recorded by using the Developer⇒Code⇒Use Relative References button. This button is a toggle. When the button appears in a different color, the macro recorder records relative references. When the button appears in the standard color, the macro recorder records absolute references. You can change the recording method at any time, even in the middle of recording.

To see how relative referencing is recorded, erase the cells in B1:G1 and then perform the following steps:

1. Activate cell B1.
2. Choose Developer⇒Code⇒Record Macro.
3. Click OK to begin recording.
4. Click the Use Relative Reference button to change the recording mode to relative.

After you click this button, it appears in a different color.

5. Enter the first six months' names in B1:G1, as in the previous example.
6. Select cell B1.
7. Stop the macro recorder.

With the recording mode set to relative, the code that Excel generates is quite different:

```
Sub Macro2()
ActiveCell.FormulaR1C1 = "Jan"
ActiveCell.Offset(0, 1).Range("A1").Select
ActiveCell.FormulaR1C1 = "Feb"
ActiveCell.Offset(0, 1).Range("A1").Select
ActiveCell.FormulaR1C1 = "Mar"
ActiveCell.Offset(0, 1).Range("A1").Select
ActiveCell.FormulaR1C1 = "Apr"
ActiveCell.Offset(0, 1).Range("A1").Select
ActiveCell.FormulaR1C1 = "May"
```

```
ActiveCell.Offset(0, 1).Range("A1").Select
ActiveCell.FormulaR1C1 = "Jun"
ActiveCell.Offset(0, -5).Range("A1").Select
End Sub
```

To test this macro, start by activating a cell other than cell B1. Then choose the Developer⇒Code⇒Macros command. Select the macro name and then click the Run button. The month names are entered beginning at the active cell.

Notice that I varied the recording procedure slightly in this example: I activated the beginning cell before I started recording. This step is important when you record macros that use the active cell as a base.

Although it looks rather complicated, this macro is actually quite simple. The first statement simply enters Jan into the active cell. (It uses the active cell because it's not preceded by a statement that selects a cell.) The next statement uses the Select method (along with the Offset property) to move the selection one cell to the right. The next statement inserts more text, and so on. Finally, the original cell is selected by calculating a relative offset rather than an absolute cell. Unlike the preceding macro, this one always starts entering text in the active cell.

**Notes:** You'll notice that this macro generates code that appears to reference cell A1 — which may seem strange because cell A1 wasn't even involved in the macro. This code is simply a byproduct of how the macro recorder works. (I discuss the Offset property later in this chapter.) At this point, all you need to know is that the macro works as it should.

The point here is that the recorder has two distinct modes, and you need to be aware of which mode you're recording in. Otherwise, the result may not be what you expected.

By the way, the code generated by Excel is more complex than it needs to be, and it's not even the most efficient way to code the operation. The macro that follows, which I entered manually, is a simpler and faster way to perform this same operation. This example demonstrates that VBA doesn't have to select a cell before it puts information into it — an important concept that can speed things up considerably.

```
Sub Macro3()
ActiveCell.Offset(0, 0) = "Jan"
ActiveCell.Offset(0, 1) = "Feb"
ActiveCell.Offset(0, 2) = "Mar"
ActiveCell.Offset(0, 3) = "Apr"
ActiveCell.Offset(0, 4) = "May"
ActiveCell.Offset(0, 5) = "Jun"
End Sub
```

In fact, this macro can be made even more efficient by using the With-End With construct:

```
Sub Macro4()
With ActiveCell
.Offset(0, 0) = "Jan"
.Offset(0, 1) = "Feb"
.Offset(0, 2) = "Mar"
.Offset(0, 3) = "Apr"

```

```
.Offset(0, 4) = "May"  
.Offset(0, 5) = "Jun"  
End With  
End Sub
```

Or, if you're a VBA guru, you can impress your colleagues by using a single statement:

```
Sub Macro5()  
ActiveCell.Resize(,6)=Array("Jan","Feb","Mar","Apr","May","Jun")  
End Sub
```

### Recording options

When you record your actions to create VBA code, you have several options in the Record Macro dialog box. The following list describes your options.

- **Macro name:** You can enter a name for the procedure that you're recording. By default, Excel uses the names Macro1, Macro2, and so on for each macro that you record. I usually just accept the default name and change the name of the procedure later. You, however, might prefer to name the macro before you record it. The choice is yours.
- **Shortcut key:** The Shortcut key option lets you execute the macro by pressing a shortcut key combination. For example, if you enter w (lowercase), you can execute the macro by pressing Ctrl+W. If you enter W (uppercase), the macro comes alive when you press Ctrl+Shift+W. Keep in mind that a shortcut key assigned to a macro overrides a built-in shortcut key (if one exists). For example, if you assign Ctrl+B to a macro, you won't be able to use the key combination to toggle the bold attribute in cells.

You can always add or change a shortcut key at any time, so you don't need to set this option while recording a macro.

- **Store Macro In:** The Store Macro In option tells Excel where to store the macro that it records. By default, Excel puts the recorded macro in a module in the active workbook. If you prefer, you can record it in a new workbook (Excel opens a blank workbook) or in your Personal Macro Workbook. (Read more about this in the sidebar, "The Personal Macro Workbook.")

**Notes:** Excel remembers your choice, so the next time you record a macro, it defaults to the same location you used previously.

- **Description:** If you like, you can enter a description for your macro in the Description box. Text you enter here appears at the beginning of your macro as a comment.

### Cleaning up recorded macros

Earlier in this chapter, you see how recording your actions while you issue a single command (the Page Layout⇒Page Setup⇒Orientation command) produces an enormous amount of VBA code. This example shows how, in many cases, the recorded code includes extraneous commands that you can delete.

The macro recorder doesn't always generate the most efficient code. If you examine the generated code, you see that Excel generally records what is selected (that is, an object) and then uses the Selection object in subsequent statements. For example, here's what is recorded if you select a range of cells

and then use some buttons on the Home tab to change the numeric formatting and apply bold and italic:

```
Range("A1:C5").Select  
Selection.Style = "Comma"  
Selection.Font.Bold = True  
Selection.Font.Italic = True
```

The recorded VBA code works, but it's just one way to perform these actions. You can also use the more efficient With-End With construct, as follows:

```
Range("A1:C5").Select  
With Selection  
.Style = "Comma"  
.Font.Bold = True  
.Font.Italic = True  
End With
```

Or you can avoid the Select method altogether and write the code even more efficiently:

```
With Range("A1:C5")  
.Style = "Comma".Font.Bold = True  
.Font.Italic = True  
End With
```

If speed is essential in your application, you always want to examine any recorded VBA code closely to make sure that it's as efficient as possible.

You, of course, need to understand VBA thoroughly before you start cleaning up your recorded macros. But for now, just be aware that recorded VBA code isn't always the best, most efficient code.

## 1.9. About Objects and Collections

If you've worked through the first part of this chapter, you have an overview of VBA, and you know the basics of working with VBA modules in the VBE. You've also seen some VBA code and were exposed to concepts like objects and properties. This section gives you additional details about objects and collections of objects.

When you work with VBA, you must understand the concept of objects and Excel's object model. It helps to think of objects in terms of a hierarchy. At the top of this model is the Application object — in this case, Excel itself. But if you're programming in VBA with Microsoft Word, the Application object is Word.

### The object hierarchy

The Application object (that is, Excel) contains other objects. Here are a few examples of objects contained in the Application object:

- Workbooks (a collection of all Workbook objects)
- Windows (a collection of all Window objects)
- AddIns (a collection of all AddIn objects)

Some objects can contain other objects. For example, the Workbooks collection consists of all open Workbook objects, and a Workbook object contains other objects, a few of which are as follows:

- Worksheets (a collection of Worksheet objects)
- Charts (a collection of Chart objects)
- Names (a collection of Name objects)

Each of these objects, in turn, can contain other objects. The Worksheets collection consists of all Worksheet objects in a Workbook. A Worksheet object contains many other objects, which include the following:

- ChartObjects (a collection of ChartObject objects)
- Range
- PageSetup
- PivotTables (a collection of PivotTable objects)

If this seems confusing, trust me, it will make sense, and you'll eventually realize that this object hierarchy setup is quite logical and well structured. By the way, the complete Excel object model is covered in the Help system.

### About collections

Another key concept in VBA programming is collections. A collection is a group of objects of the same class, and a collection is itself an object. As I note earlier, Workbooks is a collection of all Workbook objects currently open. Worksheets is a collection of all Worksheet objects contained in a particular Workbook object. You can work with an entire collection of objects or with an individual object in a collection. To reference a single object from a collection, you put the object's name or index number in parentheses after the name of the collection, like this:

*Worksheets("Sheet1")*

If Sheet1 is the first worksheet in the collection, you could also use the following reference:

*Worksheets(1)*

You refer to the second worksheet in a Workbook as Worksheets(2), and so on.

There is also a collection called Sheets, which is made up of all sheets in a workbook, whether they're worksheets or chart sheets. If Sheet1 is the first sheet in the workbook, you can reference it as follows:

*Sheets(1)*

### Referring to objects

When you refer to an object using VBA, you often must qualify the object by connecting object names with a period (also known as a dot operator). What if you had two workbooks open and they both had a worksheet named Sheet1? The solution is to qualify the reference by adding the object's container, like this:

*Workbooks("Book1").Worksheets("Sheet1")*

Without the workbook qualifier, VBA would look for Sheet1 in the active workbook.

To refer to a specific range (such as cell A1) on a worksheet named Sheet1 in a workbook named Book1, you can use the following expression:

*Workbooks("Book1").Worksheets("Sheet1").Range("A1")*

## 1.10. Properties and Methods

The fully qualified reference for the preceding example also includes the Application object, as follows:

```
Application.Workbooks("Book1").Worksheets("Sheet1").Range("A1")
```

Most of the time, however, you can omit the Application object in your references; it is assumed. If the Book1 object is the active workbook, you can even omit that object reference and use this:

```
Worksheets("Sheet1").Range("A1")
```

And — I think you know where I'm going with this — if Sheet1 is the active worksheet, you can use an even simpler expression:

```
Range("A1")
```

Simply referring to objects (as in these examples) doesn't do anything. To perform anything meaningful, you must read or modify an object's properties or specify a method to be used with an object.

It's easy to be overwhelmed with properties and methods; literally thousands are available. In this section, I describe how to access properties and methods of objects.

### Object properties

Every object has properties. For example, a Range object has a property called Value. You can write VBA code to display the Value property or write VBA code to set the Value property to a specific value. Here's a procedure that uses the VBA MsgBox function to pop up a box that displays the value in cell A1 on Sheet1 of the active workbook:

```
Sub ShowValue()  
Msgbox Worksheets("Sheet1").Range("A1").Value  
End Sub
```

**Notes: The VBA MsgBox function provides an easy way to display results while your VBA code is executing. I use it extensively throughout this book.**

The code in the preceding example displays the current setting of the Value property of a specific cell: cell A1 on a worksheet named Sheet1 in the active workbook. Note that if the active workbook doesn't have a sheet named Sheet1, the macro generates an error.

What if you want to change the Value property? The following procedure changes the value displayed in cell A1 by changing the cell's Value property:

```
Sub ChangeValue()  
Worksheets("Sheet1").Range("A1").Value = 123.45  
End Sub
```

After executing this routine, cell A1 on Sheet1 has the value 123.45.

You may want to enter these procedures into a module and experiment with them.

**Notes: Most objects have a default property. For a Range object, the default property is the Value property. Therefore, you can omit the .Value part from the preceding code, and it has the same effect. However, it's usually considered good programming practice to include the property in your code, even if it is the default property.**

The statement that follows accesses the HasFormula and the Formula properties of a Range object:

```
If Range("A1").HasFormula Then MsgBox Range("A1").Formula
```

I use an If-Then construct to display a message box conditionally: If the cell has a formula, then display the formula by accessing the Formula property. If cell A1 doesn't have a formula, nothing happens.

The Formula property is a read-write property, so you can also specify a formula by using VBA:

```
Range("D12").Formula = "=RAND()*100"
```

### Object methods

In addition to properties, objects also have methods. A method is an action that you perform with an object. Here's a simple example that uses the Clear method on a Range object. After you execute this procedure, A1:C3 on Sheet1 is empty, and all cell formatting is removed.

```
Sub ZapRange()  
Worksheets("Sheet1").Range("A1:C3").Clear  
End Sub
```

If you'd like to delete the values in a range but keep the formatting, use the ClearContents method of the Range object.

Most methods also take arguments to define the action further. Here's an example that copies cell A1 to cell B1 by using the Copy method of the Range object. In this example, the Copy method has one argument (the destination of the copy). Notice that I use the line continuation character sequence (a space followed by an underscore) in this example. You can omit the line continuation sequence and type the statement on a single line.

```
Sub CopyOne()  
Worksheets("Sheet1").Range("A1").Copy _  
Worksheets("Sheet1").Range("B1")  
End Sub
```

## 1.11. The Comment Object: A Case Study

To help you better understand the properties and methods available for an object, I focus on a particular object: the Comment object. In Excel, you create a Comment object when you choose the Review⇒Comments⇒New Comment command to enter a cell comment. In the sections that follow, you get a feel for working with objects.

### Viewing Help for the Comment object

One way to learn about a particular object is to look it up in the Help system. Figure 7-14 shows some Help topics for the Comment object. I found this Help screen by typing comment in the VBE Type a Question for Help box (to the right of the menu bar). Notice that the Help screen has a link at the bottom labeled Comment Object Members. Click that link to view the properties and methods for this object.

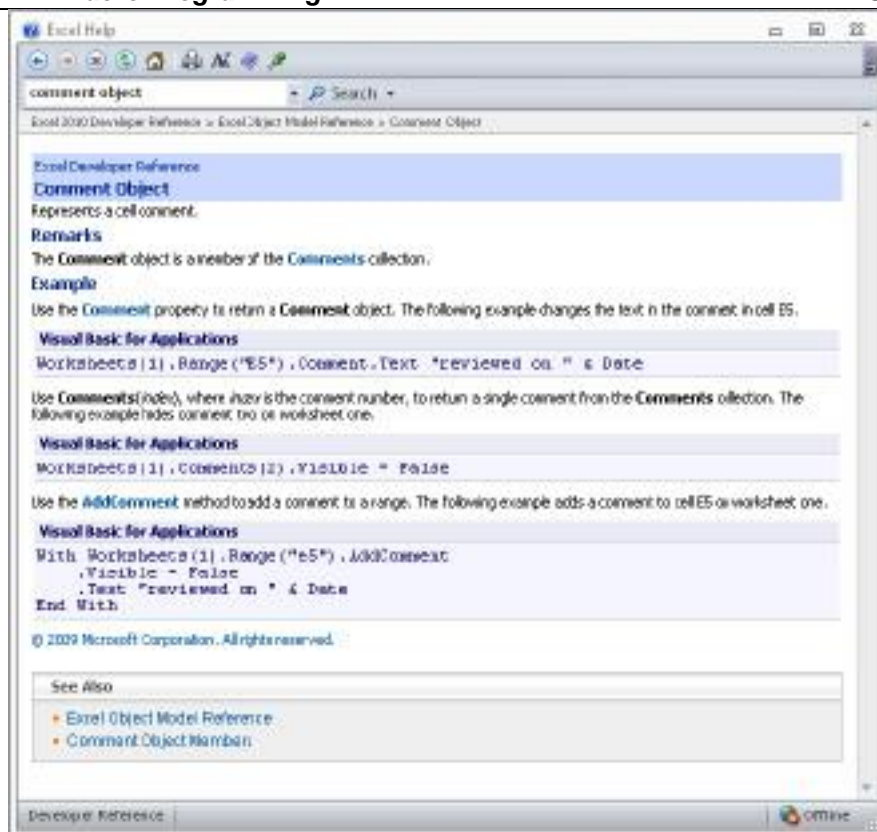


FIGURE 7-14: The main Help screen for the Comment object.

### Properties of a Comment object

The Comment object has six properties. Table 7-1 contains a list of these properties, along with a brief description of each. If a property is read-only, your VBA code can read the property but can't change it.

**Table 7-1: Properties of a Comment Object**

Property	Read-Only	Description
Application	Yes	Returns an object that represents the application that created the comment (that is, Excel).
Author	Yes	Returns the name of the person who created the comment.
Creator	Yes	Returns an integer that indicates the application in which the object was created.
Parent	Yes	Returns the parent object for the comment. (It is always a Range object.)
Shape	Yes	Returns a Shape object that represents the shape attached to the comment.
Visible	No	Is True if the comment is visible.

### Methods of a Comment object



Table 7-2 shows the methods that you can use with a Comment object. Again, these methods perform common operations that you may have performed manually with a comment at some point . . . but you probably never thought of these operations as methods.

**Table 7-2: Methods of a Comment Object**

Method	Description
Delete	Deletes a comment.
Next	Returns a Comment object that represents the next comment in the worksheet.
Previous	Returns a Comment object that represents the previous comment in the worksheet.
Text	Returns or sets the text in a comment (takes three arguments).

**Notes:** You may be surprised to see that **Text** is a method rather than a property, which leads to an important point: The distinction between properties and methods isn't always clear-cut, and the object model isn't perfectly consistent. In fact, it's not really important that you distinguish between properties and methods. As long as you get the syntax correct, it doesn't matter whether a word in your code is a property or a method.

### The Comments collection

Recall that a collection is a group of like objects. Every worksheet has a Comments collection, which consists of all Comment objects on the worksheet. If the worksheet has no comments, this collection is empty. Comments appear in the collection based on their position in the worksheet: left-to-right and then top-to-bottom.

For example, the following code refers to the first comment on Sheet1 of the active workbook:

```
Worksheets("Sheet1").Comments(1)
```

The following statement displays the text contained in the first comment on Sheet1:

```
MsgBox Worksheets("Sheet1").Comments(1).Text
```

Unlike most objects, a Comment object doesn't have a Name property. Therefore, to refer to a specific comment, you must either use an index number or (more frequently) use the Comment property of a Range object to return a specific comment.

The Comments collection is also an object and has its own set of properties and methods. For example, the Comments collection has a Count property that stores the number of items in the collection — which is the number of Comment objects in the active worksheet. The following statement displays the total number of comments on the active worksheet:

```
MsgBox ActiveSheet.Comments.Count
```

The next example shows the address of the cell that has the first comment:

```
MsgBox ActiveSheet.Comments(1).Parent.Address
```

Here, Comments(1) returns the first Comment object in the Comments collection. The Parent property of the Comment object returns its container, which is a Range object. The message box displays the Address property of the Range.

The net effect is that the statement displays the address of the cell that contains the first comment.

You can also loop through all the comments on a sheet by using the For Each-Next construct. (Looping is explained in Chapter 8.) Here's an example that displays a separate message box for each comment on the active worksheet:

```
For Each cmt in ActiveSheet.Comments
    MsgBox cmt.Text
Next cmt
```

If you'd rather not deal with a series of message boxes, use this procedure to print the comments to the Immediate window in the VBE:

```
For Each cmt in ActiveSheet.Comments
    Debug.Print cmt.Text
Next cmt
```

### About the Comment property

In this section, I've been discussing the Comment object. If you dig through the Help system, you'll find that a Range object has a property named Comment. If the cell contains a comment, the Comment property returns a Comment object. For example, the following code refers to the Comment object in cell A1:

```
Range("A1").Comment
```

If this comment were the first one on the sheet, you could refer to the same Comment object as follows:

```
ActiveSheet.Comments(1)
```

To display the comment in cell A1 in a message box, use a statement like this:

```
MsgBox Range("A1").Comment.Text
```

If cell A1 doesn't contain a comment, this statement generates an error.

### Objects within a Comment object

Working with properties is confusing at first because some properties actually return objects. Suppose that you want to determine the background color of a particular comment on Sheet1. If you look through the list of properties for a Comment object, you won't find anything that relates to color. Rather, you must do these steps:

1. Use the Comment object's Shape property to return the Shape object that's contained in the comment.
2. Use the Shape object's Fill property to return a FillFormat object.
3. Use the FillFormat object's ForeColor property to return a ColorFormat object.
4. Use the ColorFormat object's RGB property to get the color value.

Put another way, getting at the interior color for a Comment object involves accessing other objects contained in the Comment object. Here's a look at the object hierarchy that's involved:

```
Application (Excel)
    Workbook object
        Worksheet object
            Comment object
```

*Shape object*

*FillFormat object*

*ColorFormat object*

I'll be the first to admit it: This process can get very confusing! But, as an example of the elegance of VBA, you can write a single statement to change the color of a comment:

```
Worksheets("Sheet1").Comments(1).Shape.Fill.ForeColor _  
.RGB = RGB(0, 255, 0)
```

Or, if you use the *SchemeColor* property (which ranges from 0 to 80), the code is

```
Worksheets("Sheet1").Comments(1).Shape.Fill.ForeColor _  
.SchemeColor = 12
```

This type of referencing is certainly not intuitive at first, but it will eventually make sense. Fortunately, recording your actions in Excel almost always yields some insights regarding the hierarchy of the objects involved.

By the way, to change the color of the text in a comment, you need to access the *Comment* object's *TextFrame* object, which contains the *Characters* object, which contains the *Font* object. Then you have access to the *Font* object's *Color* or *ColorIndex* properties. Here's an example that sets the *ColorIndex* property to 5:

```
Worksheets("Sheet1").Comments(1) _  
.Shape.TextFrame.Characters.Font.ColorIndex = 5
```

### **Determining whether a cell has a comment**

The following statement displays the comment in cell A1 of the active sheet:

```
MsgBox Range("A1").Comment.Text
```

If cell A1 doesn't have a comment, executing this statement generates a cryptic error message: Object variable or With block variable not set.

To determine whether a particular cell has a comment, you can write code to check whether the *Comment* object is *Nothing*. (Yes, *Nothing* is a valid keyword.) The following statement displays *True* if cell A1 doesn't have a comment:

```
MsgBox Range("A1").Comment Is Nothing
```

Note that I use the *Is* keyword and not an equal sign.

You can take this one step further and write a statement that displays the cell comment only if the cell actually has a comment (and does not generate an error if the cell lacks a comment). The statement that follows accomplishes this task:

```
If Not Range("A1").Comment Is Nothing Then _  
MsgBox Range("A1").Comment.Text
```

Notice that I used the *Not* keyword, which negates the *True* value that's returned if the cell has no comment. The statement, in essence, uses a double-negative to test a condition: If the comment isn't nothing, then display it. If this statement is confusing, think about it for a while, and it will make sense.

### **Adding a new Comment object**

You may have noticed that the list of methods for the *Comment* object doesn't include a method to add a new comment. This is because the *AddComment* method belongs to the *Range* object. The following statement adds a comment (an empty comment) to cell A1 on the active worksheet:

## 1.12. Some Useful Application Properties

*Range("A1").AddComment*

If you consult the Help system, you discover that the AddComment method takes an argument that represents the text for the comment. Therefore, you can add a comment and then add text to the comment with a single statement:

*Range("A1").AddComment "Formula developed by JW."*

When you're working with Excel, only one workbook at a time can be active. And if the sheet is a worksheet, one cell is the active cell (even if a multicell range is selected). VBA knows about active workbooks, worksheets, and cells and lets you refer to these active objects in a simplified manner. This method of referring to objects is often useful because you won't always know the exact workbook, worksheet, or range that you want to operate on. VBA makes it easy by providing properties of the Application object. For example, the Application object has an ActiveCell property that returns a reference to the active cell. The following instruction assigns the value 1 to the active cell:

*ActiveCell.Value = 1*

Notice that I omitted the reference to the Application object in the preceding example because it's assumed. It's important to understand that this instruction will fail if the active sheet isn't a worksheet. For example, if VBA executes this statement when a chart sheet is active, the procedure halts, and you get an error message.

If a range is selected in a worksheet, the active cell is a cell within the selected range. In other words, the active cell is always a single cell (never a multicell range).

The Application object also has a Selection property that returns a reference to whatever is selected, which may be a single cell (the active cell), a range of cells, or an object such as ChartObject, TextBox, or Shape.

Table 7-3 lists the other Application properties that are useful when working with cells and ranges.

**Table 7-3: Some Useful Properties of the Application Object**

Property	Object Returned
ActiveCell	The active cell.
ActiveChart	The active chart sheet or chart contained in a ChartObject on a worksheet. This property is Nothing if a chart isn't active.
ActiveSheet	The active sheet (worksheet or chart).
ActiveWindow	The active window.
ActiveWorkbook	The active workbook.
Selection	The object selected. (It could be a Range object, Shape, ChartObject, and so on.)
ThisWorkbook	The workbook that contains the VBA procedure being executed. This object may or may not be the same as the ActiveWorkbook object.

The advantage of using these properties to return an object is that you don't need to know which cell, worksheet, or workbook is active, and you don't need to provide a specific reference to it. This allows you to write VBA code that isn't specific to a particular workbook, sheet, or range. For example, the following

instruction clears the contents of the active cell, even though the address of the active cell isn't known:

*ActiveCell.ClearContents*

The example that follows displays a message that tells you the name of the active sheet:

*MsgBox ActiveSheet.Name*

If you want to know the name and directory path of the active workbook, use a statement like this:

*MsgBox ActiveWorkbook.FullName*

If a range on a worksheet is selected, you can fill the entire range with a value by executing a single statement. In the following example, the Selection property of the Application object returns a Range object that corresponds to the selected cells. The instruction simply modifies the Value property of this Range object, and the result is a range filled with a single value:

*Selection.Value = 12*

Note that if something other than a range is selected (such as a ChartObject or a Shape), the preceding statement generates an error because ChartObject and Shape objects don't have a Value property.

The following statement, however, enters a value of 12 into the Range object that was selected before a non-Range object was selected. If you look up the RangeSelection property in the Help system, you find that this property applies only to a Window object.

*ActiveWindow.RangeSelection.Value = 12*

To find out how many cells are selected in the active window, access the Count property. Here's an example:

*MsgBox ActiveWindow.RangeSelection.Count*

### 1.13. Working with Range Objects

Much of the work that you will do in VBA involves cells and ranges in worksheets. The earlier discussion on relative versus absolute macro recording (see "Relative or absolute recording?") exposes you to working with cells in VBA, but you need to know a lot more.

A Range object is contained in a Worksheet object and consists of a single cell or range of cells on a single worksheet. In the sections that follow, I discuss three ways of referring to Range objects in your VBA code:

- The Range property of a Worksheet or Range class object
- The Cells property of a Worksheet object
- The Offset property of a Range object

#### The Range property

The Range property returns a Range object. If you consult the Help system for the Range property, you learn that this property has two syntaxes:

*object.Range(cell1)*

*object.Range(cell1, cell2)*

The Range property applies to two types of objects: a Worksheet object or a Range object. Here, cell1 and cell2 refer to placeholders for terms that Excel recognizes as identifying the range (in the first instance) and delineating the

range (in the second instance). Following are a few examples of using the Range property.

You've already seen examples like the following one earlier in the chapter. The instruction that follows simply enters a value into the specified cell. In this case, it puts the value 12.3 into cell A1 on Sheet1 of the active workbook:

```
Worksheets("Sheet1").Range("A1").Value = 12.3
```

The Range property also recognizes defined names in workbooks. Therefore, if a cell is named Input, you can use the following statement to enter a value into that named cell:

```
Worksheets("Sheet1").Range("Input").Value = 100
```

The example that follows enters the same value into a range of 20 cells on the active sheet. If the active sheet isn't a worksheet, the statement causes an error message:

```
ActiveSheet.Range("A1:B10").Value = 2
```

Working with merged cells

The next example produces exactly the same result as the preceding example:

```
Range("A1", "B10") = 2
```

The sheet reference is omitted, however, so the active sheet is assumed. Also, the value property is omitted, so the default property (which is Value for a Range object) is assumed. This example also uses the second syntax of the Range property. With this syntax, the first argument is the cell at the top left of the range, and the second argument is the cell at the lower right of the range.

The following example uses the Excel range intersection operator (a space) to return the intersection of two ranges. In this case, the intersection is a single cell, C6. Therefore, this statement enters 3 into cell C6:

```
Range("C1:C10 A6:E6") = 3
```

And finally, this next example enters the value 4 into five cells: that is, a noncontiguous range. The comma serves as the union operator.

```
Range("A1,A3,A5,A7,A9") = 4
```

So far, all the examples have used the Range property on a Worksheet object. As I mentioned, you can also use the Range property on a Range object. This concept can be rather confusing, but bear with me.

Following is an example of using the Range property on a Range object. (In this case, the Range object is the active cell.) This example treats the Range object as if it were the upper-left cell in the worksheet, and then it enters a value of 5 into the cell that would be B2. In other words, the reference returned is relative to the upper-left corner of the Range object. Therefore, the statement that follows enters a value of 5 into the cell directly to the right and one row below the active cell:

```
ActiveCell.Range("B2") = 5
```

I said this is confusing. Fortunately, you can access a cell relative to a range in a much clearer way: the Offset property. I discuss this property after the next section.

### The Cells property

Another way to reference a range is to use the Cells property. You can use the Cells property, like the Range property, on Worksheet objects and Range

objects. Check the Help system, and you see that the Cells property has three syntaxes:

*object.Cells(rowIndex, columnIndex)*

*object.Cells(rowIndex)*

*object.Cells*

Some examples demonstrate how to use the Cells property. The first example enters the value 9 into cell A1 on Sheet1. In this case, I'm using the first syntax, which accepts the index number of the row (from 1 to 1048576) and the index number of the column (from 1 to 16384):

*Worksheets("Sheet1").Cells(1, 1) = 9*

Here's an example that enters the value 7 into cell D3 (that is, row 3, column 4) in the active worksheet:

*ActiveSheet.Cells(3, 4) = 7*

You can also use the Cells property on a Range object. When you do so, the Range object returned by the Cells property is relative to the upper-left cell of the referenced Range. Confusing? Probably. An example may help clear up any confusion. The following instruction enters the value 5 into the active cell. Remember, in this case, the active cell is treated as if it were cell A1 in the worksheet:

*ActiveCell.Cells(1, 1) = 5*

To enter a value of 5 into the cell directly below the active cell, you can use the following instruction:

*ActiveCell.Cells(2, 1) = 5*

Think of the preceding example as though it said this: "Start with the active cell and consider this cell as cell A1. Place 5 in the cell in the second row and the first column."

The second syntax of the Cells method uses a single argument that can range from 1 to 17,179,869,184. This number is equal to the number of cells in an Excel 2010 worksheet. The cells are numbered starting from A1 and continuing right and then down to the next row. The 16,384th cell is XFD1; the 16,385th is A2.

The next example enters the value 2 into cell SZ1 (which is the 520th cell in the worksheet) of the active worksheet:

*ActiveSheet.Cells(520) = 2*

To display the value in the last cell in a worksheet (XFD1048576), use this statement:

*MsgBox ActiveSheet.Cells(17179869184)*

You can also use this syntax with a Range object. In this case, the cell returned is relative to the Range object referenced. For example, if the Range object is A1:D10 (40 cells), the Cells property can have an argument from 1 to 40 and can return one of the cells in the Range object. In the following example, a value of 2000 is entered into cell A2 because A2 is the fifth cell (counting from the top, to the right, and then down) in the referenced range:

*Range("A1:D10").Cells(5) = 2000*

*Range("A1:D10").Cells(41)=2000*

The third syntax for the Cells property simply returns all cells on the referenced worksheet. Unlike the other two syntaxes, in this one, the return data isn't a

single cell. This example uses the ClearContents method on the range returned by using the Cells property on the active worksheet. The result is that the content of every cell on the worksheet is cleared:

```
ActiveSheet.Cells.ClearContents
```

### The Offset property

The Offset property, like the Range and Cells properties, also returns a Range object. But unlike the other two methods that I discussed, the Offset property applies only to a Range object and no other class. Its syntax is as follows:

```
object.Offset(rowOffset, columnOffset)
```

The Offset property takes two arguments that correspond to the relative position from the upper-left cell of the specified Range object. The arguments can be positive (down or to the right), negative (up or to the left), or zero. The example that follows enters a value of 12 into the cell directly below the active cell:

```
ActiveCell.Offset(1,0).Value = 12
```

The next example enters a value of 15 into the cell directly above the active cell:

```
ActiveCell.Offset(-1,0).Value = 15
```

If the active cell is in row 1, the Offset property in the preceding example generates an error because it can't return a Range object that doesn't exist.

The Offset property is quite useful, especially when you use variables within looping procedures. I discuss these topics in the next chapter.

When you record a macro using the relative reference mode, Excel uses the Offset property to reference cells relative to the starting position (that is, the active cell when macro recording begins). For example, I used the macro recorder to generate the following code. I started with the cell pointer in cell B1, entered values into B1:B3, and then returned to B1.

```
Sub Macro1()  
    ActiveCell.FormulaR1C1 = "1"  
    ActiveCell.Offset(1, 0).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "2"  
    ActiveCell.Offset(1, 0).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "3"  
    ActiveCell.Offset(-2, 0).Range("A1").Select  
End Sub
```

Notice that the macro recorder uses the FormulaR1C1 property. Normally, you want to use the Value property to enter a value into a cell. However, using FormulaR1C1 or even Formula produces the same result.

Also notice that the generated code references cell A1 — a cell that was even involved in the macro. This notation is a quirk in the macro recording procedure that makes the code more complex than necessary. You can delete all references to Range("A1"), and the macro still works perfectly:

```
Sub Modified_Macro1()  
    ActiveCell.FormulaR1C1 = "1"  
    ActiveCell.Offset(1, 0).Select  
    ActiveCell.FormulaR1C1 = "2"  
    ActiveCell.Offset(1, 0).Select
```



### 1.14. Things to Know about Objects

```
ActiveCell.FormulaR1C1 = "3"
```

```
ActiveCell.Offset(-2, 0).Select
```

```
End Sub
```

In fact, here's a much more efficient version of the macro (which I wrote myself) that doesn't do any selecting:

```
Sub Macro1()
```

```
ActiveCell = 1
```

```
ActiveCell.Offset(1, 0) = 2
```

```
ActiveCell.Offset(2, 0) = 3
```

```
End Sub
```

The preceding sections introduced you to objects (including collections), properties, and methods. But I've barely scratched the surface.

#### Essential concepts to remember

In this section, I note some additional concepts that are essential for would-be VBA gurus. These concepts become clearer when you work with VBA and read subsequent chapters:

- Objects have unique properties and methods. Each object has its own set of properties and methods. Some objects, however, share some properties (for example, Name) and some methods (such as Delete).
- You can manipulate objects without selecting them. This idea may be contrary to how you normally think about manipulating objects in Excel. The fact is that it's usually more efficient to perform actions on objects without selecting them first. When you record a macro, Excel generally selects the object first. This step isn't necessary and may actually make your macro run more slowly.
- It's important that you understand the concept of collections. Most of the time, you refer to an object indirectly by referring to the collection that it's in. For example, to access a Workbook object named Myfile, reference the Workbooks collection as follows:

```
Workbooks("Myfile.xlsx")
```

This reference returns an object, which is the workbook with which you're concerned.

Properties can return a reference to another object. For example, in the following statement, the Font property returns a Font object contained in a Range object. Bold is a property of the Font object, not the Range object.

```
Range("A1").Font.Bold = True
```

- You can refer to the same object in many different ways. Assume that you have a workbook named Sales, and it's the only workbook open. Then assume that this workbook has one worksheet, named Summary. You can refer to the sheet in any of the following ways:

```
Workbooks("Sales.xlsx").Worksheets("Summary")
```

```
Workbooks(1).Worksheets(1)
```

```
Workbooks(1).Sheets(1)
```

```
Application.ActiveWorkbook.ActiveSheet
```

```
ActiveWorkbook.ActiveSheet
```

*ActiveSheet*

The method that you use is usually determined by how much you know about the workspace. For example, if more than one workbook is open, the second and third methods aren't reliable. If you want to work with the active sheet (whatever it may be), any of the last three methods would work. To be absolutely sure that you're referring to a specific sheet on a specific workbook, the first method is your best choice.

**Learning more about objects and properties**

If this is your first exposure to VBA, you're probably a bit overwhelmed by objects, properties, and methods. I don't blame you. If you try to access a property that an object doesn't have, you get a runtime error, and your VBA code grinds to a screeching halt until you correct the problem.

Fortunately, there are several good ways to learn about objects, properties, and methods.

**Read the rest of the book**

Don't forget, the name of this chapter is "Introducing Visual Basic for Applications." The remainder of this book covers many additional details and provides many useful and informative examples.

**Record your actions**

The absolute best way to become familiar with VBA, without question, is to simply turn on the macro recorder and record some actions that you perform in Excel. This approach is a quick way to learn the relevant objects, properties, and methods for a task. It's even better if the VBA module in which the code is being recorded is visible while you're recording.

**Use the Help system**

The main source of detailed information about Excel's objects, methods, and procedures is the Help system. Many people forget about this resource.

**Use the Object Browser**

The Object Browser is a handy tool that lists every property and method for every object available. When the VBE is active, you can bring up the Object Browser in any of the following three ways:

- Press F2.
- Choose the View⇒Object Browser command from the menu.
- Click the Object Browser tool on the Standard toolbar.

The Object Browser is shown in Figure 7-15.

The drop-down list in the upper-left corner of the Object Browser includes a list of all object libraries that you have access to:

- Excel itself
- MSForms (used to create custom dialog boxes)
- Office (objects common to all Microsoft Office applications)

- Stdole (OLE automation objects)
- VBA
- The current project (the project that's selected in the Project Explorer) and any workbooks referenced by that project

Your selection in this upper-left drop-down list determines what is displayed in the Classes window, and your selection in the Classes window determines what is visible in the Members Of panel.

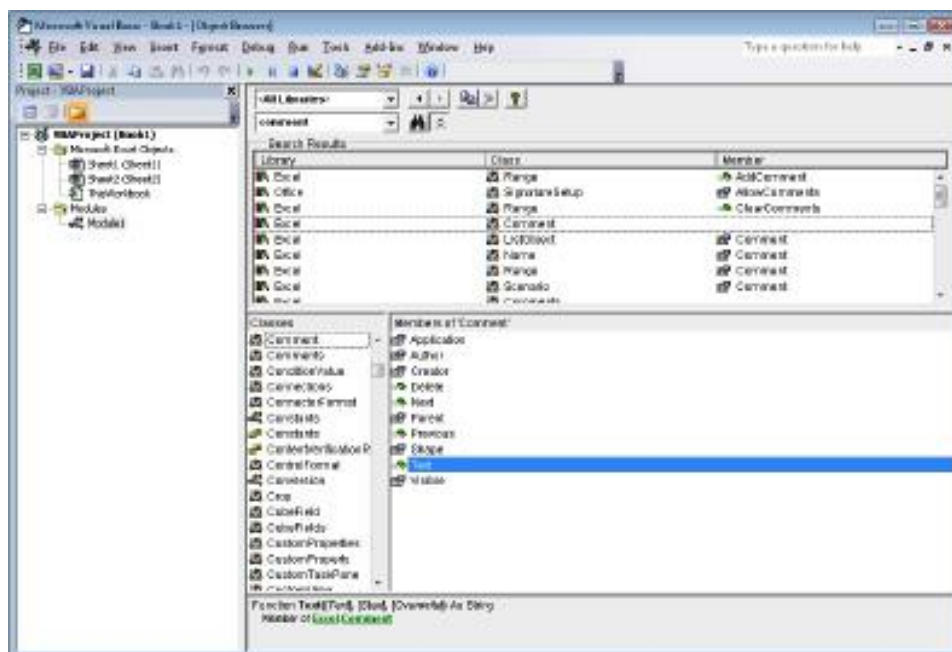


FIGURE 7-15: The Object Browser is a great reference source.

After you select a library, you can search for a particular text string to get a list of properties and methods that contain the text. You do so by entering the text in the second drop-down list and then clicking the binoculars (Search) icon. For example, assume that you're working on a project that manipulates cell comments:

1. Select the library of interest.

If you're not sure which object library is appropriate, you can select <All Libraries>.

2. Enter Comment in the drop-down list below the library list.
3. Click the binoculars icon to begin the text search.

The Search Results window displays the matching text. Select an object to display its classes in the Classes window. Select a class to display its members (properties, methods, and constants). Pay attention to the bottom pane, which shows more information about the object. You can press F1 to go directly to the appropriate help topic.

The Object Browser may seem complex at first, but its usefulness to you will increase over time.

### Experiment with the Immediate window

As I describe in the sidebar earlier in this chapter (see “About the code examples”), the Immediate window of the VBE is very useful for testing statements and trying out various VBA expressions. I generally keep the Immediate window visible at all times, and I use it frequently to test various expressions and to help in debugging code.

## 2.1. VBA Language Elements: An Overview

# 2. VBA Programming Fundamentals

If you've used other programming languages, much of the information in this chapter may sound familiar. However, VBA has a few unique wrinkles, so even experienced programmers may find some new information.

In Chapter 7, I present an overview of objects, properties, and methods, but I don't tell you much about how to manipulate objects so that they do meaningful things. This chapter gently nudges you in that direction by exploring the VBA language elements, which are the keywords and control structures that you use to write VBA routines.

To get the ball rolling, I start by presenting a simple VBA Sub procedure. The following code, which is stored in a VBA module, calculates the sum of the first 100 positive integers. When the code finishes executing, the procedure displays a message with the result.

```
Sub VBA_Demo()  
    ' This is a simple VBA Example  
    Dim Total As Long, i As Long  
    Total = 0  
    For i = 1 To 100  
        Total = Total + i  
    Next i  
    MsgBox Total  
End Sub
```

This procedure uses some common VBA language elements, including:

- A comment (the line that begins with an apostrophe)
- A variable declaration statement (the line that begins with Dim)
- Two variables (Total and i)
- Two assignment statements (Total = 0 and Total = Total + i)
- A looping structure (For-Next)
- A VBA function (MsgBox)

All these language elements are discussed in subsequent sections of this chapter.

**Notes: VBA procedures need not manipulate any objects. The preceding procedure, for example, doesn't do anything with objects. It simply works with numbers.**

### Entering VBA code

VBA code, which resides in a VBA module, consists of instructions. The accepted practice is to use one instruction per line. This standard isn't a requirement, however; you can use a colon to separate multiple instructions on a single line. The following example combines four instructions on one line:

```
Sub OneLine()
```

```
x= 1: y= 2: z= 3: MsgBox x + y + z
```

```
End Sub
```

Most programmers agree that code is easier to read if you use one instruction per line:

```
Sub MultipleLines()
```

```
x = 1
```

```
y = 2
```

```
z = 3
```

```
MsgBox x + y + z
```

```
End Sub
```

Each line can be as long as you like; the VBA module window scrolls to the left when you reach the right side. For lengthy lines, you may want to use VBA's line continuation sequence: a space followed by an underscore (\_). For example:

```
Sub LongLine()
```

```
SummedValue = _
```

```
Worksheets("Sheet1").Range("A1").Value + _
```

```
Worksheets("Sheet2").Range("A1").Value
```

```
End Sub
```

When you record macros, Excel often uses underscores to break long statements into multiple lines.

After you enter an instruction, VBA performs the following actions to improve readability:

- It inserts spaces between operators. If you enter `Ans=1+2` (without spaces), for example, VBA converts it to

```
Ans = 1 + 2
```

- It adjusts the case of the letters for keywords, properties, and methods. If you enter the following text: `Result=activesheet.range("a1").value=12`

VBA converts it to

```
Result = ActiveSheet.Range("a1").Value = 12
```

Notice that text within quotation marks (in this case, "a1") isn't changed.

- Because VBA variable names aren't case-sensitive, the interpreter by default adjusts the names of all variables with the same letters so that their case matches the case of letters that you most recently typed. For example, if you first specify a variable as `myvalue` (all lowercase) and then enter the variable as `MyValue` (mixed case), VBA changes all other occurrences of the variable to `MyValue`. An exception occurs if you declare the variable with `Dim` or a similar statement; in this case, the variable name always appears as it was declared.
- VBA scans the instruction for syntax errors. If VBA finds an error, it changes the color of the line and might display a message describing the problem. Choose the Visual Basic Editor Tools⇒Options command to display the Options dialog box, where you control the error color (use the Editor Format tab) and whether the error message is displayed (use the Auto Syntax Check option in the Editor tab).

## 2.2. Comments

A comment is descriptive text embedded within your code and ignored by VBA. It's a good idea to use comments liberally to describe what you're doing because an instruction's purpose isn't always obvious.

You can use a complete line for your comment, or you can insert a comment after an instruction on the same line. A comment is indicated by an apostrophe. VBA ignores any text that follows an apostrophe — except when the apostrophe is contained within quotation marks — up until the end of the line. For example, the following statement doesn't contain a comment, even though it has an apostrophe:

```
Msg = "Can't continue"
```

The following example shows a VBA procedure with three comments:

```
Sub CommentDemo()  
    ' This procedure does nothing of value  
    x = 0 'x represents nothingness  
    ' Display the result  
    MsgBox x  
End Sub
```

Although the apostrophe is the preferred comment indicator, you can also use the Rem keyword to mark a line as a comment. For example:

```
Rem -- The next statement prompts the user for a filename
```

The Rem keyword (short for Remark) is essentially a holdover from older versions of BASIC and is included in VBA for the sake of compatibility. Unlike the apostrophe, Rem can be written only at the beginning of a line, not on the same line as another instruction.

**Notes: Using comments is definitely a good idea, but not all comments are equally beneficial. To be useful, comments should convey information that's not immediately obvious from reading the code. Otherwise, you're just chewing up valuable bytes and increasing the size of your workbook.**

Following are a few general tips on making the best use of comments:

- Use comments to describe briefly the purpose of each procedure that you write.
- Use comments to describe changes that you make to a procedure.
- Use comments to indicate that you're using functions or constructs in an unusual or nonstandard manner.
- Use comments to describe the purpose of variables so that you and other people can decipher otherwise cryptic names.
- Use comments to describe workarounds that you develop to overcome Excel bugs or limitations.
- Write comments while you code rather than after.

**Notes: In some cases, you may want to test a procedure without including a particular instruction or group of instructions. Instead of deleting the instruction, simply turn it into a comment by inserting an apostrophe at the beginning. VBA then ignores the instruction(s) when the routine is**

### 2.3. Variables, Data Types, and Constants

executed. To convert the comment back to an instruction, just delete the apostrophe.

The Visual Basic Editor (VBE) Edit toolbar contains two very useful buttons. (The Edit toolbar isn't displayed by default. To display this toolbar, choose View⇒Toolbars⇒Edit.) Select a group of instructions and then click the Comment Block button to convert the instructions to comments. The Uncomment Block button converts a group of comments back to instructions.

VBA's main purpose in life is to manipulate data. Some data resides in objects, such as worksheet ranges. Other data is stored in variables that you create.

A variable is simply a named storage location in your computer's memory. Variables can accommodate a wide variety of data types — from simple Boolean values (True or False) to large, double-precision values (see the following section). You assign a value to a variable by using the equal sign operator (more about this process in the upcoming section, "Assignment Statements").

You make your life easier if you get into the habit of making your variable names as descriptive as possible. VBA does, however, have a few rules regarding variable names:

- You can use alphabetic characters, numbers, and some punctuation characters, but the first character must be alphabetic.
- VBA doesn't distinguish between case. To make variable names more readable, programmers often use mixed case (for example, InterestRate rather than interestrate).
- You can't use spaces or periods. To make variable names more readable, programmers often use the underscore character (Interest\_Rate).
- You can't embed special type declaration characters (#, \$, %, &, or !) in a variable name.
- Variable names can be as long as 254 characters — but using such long variable names isn't recommended.

The following list contains some examples of assignment expressions that use various types of variables. The variable names are to the left of the equal sign. Each statement assigns the value to the right of the equal sign to the variable on the left.

*x = 1*

*InterestRate = 0.075*

*LoanPayoffAmount = 243089.87*

*DataEntered = False*

*x = x + 1*

*MyNum = YourNum \* 1.25*

*UserName = "Bob Johnson"*

*DateStarted = #12/14/2009#*

VBA has many reserved words, which are words that you can't use for variable or procedure names. If you attempt to use one of these words, you get an error message. For example, although the reserved word Next might make a very descriptive variable name, the following instruction generates a syntax error:

*Next = 132*

Unfortunately, syntax error messages aren't always descriptive. If the Auto Syntax Check option is turned on you get the error: Compile error: Expected: variable. If Auto Syntax Check is turned off, attempting to execute this statement results in: Compile error: Syntax error. It would be more helpful if the error message were something like Reserved word used as a variable. So if an instruction produces a strange error message, check the VBA Help system to ensure that your variable name doesn't have a special use in VBA.

### Defining data types

VBA makes life easy for programmers because it can automatically handle all the details involved in dealing with data. Not all programming languages make it so easy. For example, some languages are strictly typed, which means that the programmer must explicitly define the data type for every variable used.

Data type refers to how data is stored in memory — as integers, real numbers, strings, and so on. Although VBA can take care of data typing automatically, it does so at a cost: slower execution and less efficient use of memory. As a result, letting VBA handle data typing may present problems when you're running large or complex applications. Another advantage of explicitly declaring your variables as a particular data type is that VBA can perform some additional error checking at the compile stage. These errors might otherwise be difficult to locate.

Table 8-1 lists VBA's assortment of built-in data types. (Note that you can also define custom data types, which I describe later in this chapter in “User-Defined Data Types.”)

**Table 8-1: VBA Built-In Data Types**

Data Type	Bytes Used	Range of Values
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	–32,768 to 32,767
Long	4 bytes	–2,147,483,648 to 2,147,483,647
Single	4 bytes	–3.402823E38 to –1.401298E-45 (for negative values); 1.401298E-45 to 3.402823E38 (for positive values)
Double	8 bytes	–1.79769313486232E308 to –4.94065645841247E-324 (negative values); 4.94065645841247E-324 to 1.79769313486232E308 (for positive values)
Currency	8 bytes	–922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point; +/-7.9228162514264337593543950335 with 28 places to the right of the decimal
Date	8 bytes	January 1, 0100 to December 31, 9999
Object	4 bytes	Any object reference
String (variable length)	10 bytes + string length	0 to approximately 2 billion characters



String (fixed length)	Length of string	1 to approximately 65,400 characters
Variant (with numbers)	16 bytes	Any numeric value up to the range of a double data type. It can also hold special values, such as Empty, Error, Nothing, and Null.
Variant (with characters)	22 bytes + string length	0 to approximately 2 billion
User-defined	Varies	Varies by element

**Notes:** The Decimal data type is rather unusual because you can't actually declare it. In fact, it is a subtype of a variant. You need to use the VBA CDec function to convert a variant to the Decimal data type.

Generally, it's best to use the data type that uses the smallest number of bytes yet still can handle all the data that will be assigned to it. When VBA works with data, execution speed is partially a function of the number of bytes that VBA has at its disposal. In other words, the fewer bytes used by data, the faster VBA can access and manipulate the data.

For worksheet calculation, Excel uses the Double data type, so that's a good choice for processing numbers in VBA when you don't want to lose any precision. For integer calculations, you can use the Integer type (which is limited to values less than or equal to 32,767). Otherwise, use the Long data type. In fact, using the Long data type even for values less than 32,767 is recommended, because this data type may be a bit faster than using the Integer type. When dealing with Excel worksheet row numbers, you want to use the Long data type because the number of rows in a worksheet exceeds the maximum value for the Integer data type.

### Benchmarking variant data types

To test whether data typing is important, I developed the following routine, which performs more than 300 million meaningless calculations in a loop and then displays the procedure's total execution time:

```

Sub TimeTest()
    Dim x As Long, y As Long
    Dim A As Double, B As Double, C As Double
    Dim i As Long, j As Long
    Dim StartTime As Date, EndTime As Date
    ' Store the starting time
    StartTime = Timer
    ' Perform some calculations
    x = 0
    y = 0
    For i = 1 To 10000
        x = x + 1
        y = x + 1
    For j = 1 To 10000

```

```
A = x + y + i
B = y - x - i
C = x / y * i
Next j
Next i
' Get ending time
EndTime = Timer
' Display total time in seconds
MsgBox Format(EndTime - StartTime, "0.0")
End Sub
```

On my system, this routine took 7.7 seconds to run. (The time will vary, depending on your system's processor speed.) I then commented out the Dim statements, which declare the data types. That is, I turned the Dim statements into comments by adding an apostrophe at the beginning of the lines. As a result, VBA used the default data type, Variant. I ran the procedure again. It took 25.4 seconds, more than three times as long as before.

The moral is simple: If you want your VBA applications to run as fast as possible, declare your variables!

A workbook that contains this code is available on the companion CD-ROM in a file named timing text.xlsm.

### Declaring variables

If you don't declare the data type for a variable that you use in a VBA routine, VBA uses the default data type, Variant. Data stored as a Variant acts like a chameleon: It changes type, depending on what you do with it.

The following procedure demonstrates how a variable can assume different data types:

```
Sub VariantDemo()
MyVar = "123"
MyVar = MyVar / 2
MyVar = "Answer: " & MyVar
MsgBox MyVar
End Sub
```

In the VariantDemo procedure, MyVar starts out as a three-character string. Then this string is divided by two and becomes a numeric data type. Next, MyVar is appended to a string, converting MyVar back to a string. The MsgBox statement displays the final string: Answer: 61.5.

To further demonstrate the potential problems in dealing with Variant data types, try executing this procedure:

```
Sub VariantDemo2()
MyVar = "123"
MyVar = MyVar + MyVar
MyVar = "Answer: " & MyVar
```

```
MsgBox MyVar  
End Sub
```

The message box displays Answer: 123123. This is probably not what you wanted. When dealing with variants that contain text strings, the + operator performs string concatenation.

### Determining a data type

You can use the VBA TypeName function to determine the data type of a variable. Here's a modified version of the previous procedure. This version displays the data type of MyVar at each step. You see that it starts out as a string, is then converted to a double, and finally ends up as a string again.

```
Sub VariantDemo2()  
MyVar = "123"  
MsgBox TypeName(MyVar)  
MyVar = MyVar / 2  
MsgBox TypeName(MyVar)  
MyVar = "Answer: " & MyVar  
MsgBox TypeName(MyVar)  
MsgBox MyVar  
End Sub
```

Thanks to VBA, the data type conversion of undeclared variables is automatic. This process may seem like an easy way out, but remember that you sacrifice speed and memory — and you run the risk of errors that you may not even know about.

Declaring each variable in a procedure before you use it is an excellent habit. Declaring a variable tells VBA its name and data type. Declaring variables provides two main benefits:

- Your programs run faster and use memory more efficiently. The default data type, Variant, causes VBA to repeatedly perform time-consuming checks and reserve more memory than necessary. If VBA knows the data type, it doesn't have to investigate, and it can reserve just enough memory to store the data.
- You avoid problems involving misspelled variable names. This benefit assumes that you use Option Explicit to force yourself to declare all variables (see the next section). Say that you use an undeclared variable named CurrentRate. At some point in your routine, however, you insert the statement CurentRate = .075. This misspelled variable name, which is very difficult to spot, will likely cause your routine to give incorrect results.

### Forcing yourself to declare all variables

To force yourself to declare all the variables that you use, include the following as the first instruction in your VBA module:

```
Option Explicit
```

When this statement is present, VBA won't even execute a procedure if it contains an undeclared variable name. VBA issues the error message shown in Figure 8-1, and you must declare the variable before you can proceed.



FIGURE 8-1: VBA's way of telling you that your procedure contains an undeclared variable.

**Notes:** To ensure that the Option Explicit statement is inserted automatically whenever you insert a new VBA module, enable the Require Variable Declaration option in the Editor tab of the VBE Options dialog box (choose Tools⇒Options). I highly recommend doing so. Be aware, however, that this option doesn't affect existing modules.

### Scoping variables

A variable's scope determines in which modules and procedures you can use the variable. Table 8-2 lists the three ways in which a variable can be scoped.

**Table 8-2: Variable Scope**

Scope	How a Variable with This Scope Is Declared
Single procedure	Include a Dim or Static statement within the procedure.
Single module	Include a Dim or Private statement before the first procedure in a module.
All modules	Include a Public statement before the first procedure in a module.

### Local variables

A local variable is a variable declared within a procedure. You can use local variables only in the procedure in which they're declared. When the procedure ends, the variable no longer exists, and Excel frees up the memory that the variable used. If you need the variable to retain its value when the procedure ends, declare it as a Static variable. (See "Static variables," later in this section.)

The most common way to declare a local variable is to place a Dim statement between a Sub statement and an End Sub statement. Dim statements usually are placed right after the Sub statement, before the procedure's code.

The following procedure uses six local variables declared by using Dim statements:

```
Sub MySub()
    Dim x As Integer
```

```

Dim First As Long
Dim InterestRate As Single
Dim TodaysDate As Date
Dim UserName As String
Dim MyValue
' - [The procedure's code goes here] -
End Sub

```

Notice that the last Dim statement in the preceding example doesn't declare a data type; it simply names the variable. As a result, that variable becomes a variant.

You also can declare several variables with a single Dim statement. For example:

```

Dim x As Integer, y As Integer, z As Integer
Dim First As Long, Last As Double

```

**Notes:** Unlike some languages, VBA doesn't let you declare a group of variables to be a particular data type by separating the variables with commas. For example, the following statement, although valid, does not declare all the variables as integers:

```
Dim i, j, k As Integer
```

**In VBA, only k is declared to be an integer; the other variables are declared variants. To declare i, j, and k as integers, use this statement:**

```
Dim i As Integer, j As Integer, k As Integer
```

If a variable is declared with a local scope, other procedures in the same module can use the same variable name, but each instance of the variable is unique to its own procedure.

In general, local variables are the most efficient because VBA frees up the memory that they use when the procedure ends.

### Module-wide variables

Sometimes, you want a variable to be available to all procedures in a module. If so, just declare the variable before the module's first procedure (outside of any procedures or functions).

### Another way of data-typing variables

Like most other dialects of BASIC, VBA lets you append a character to a variable's name to indicate the data type. For example, you can declare the MyVar variable as an integer by tacking % onto the name:

```
Dim MyVar%
```

Type-declaration characters exist for most VBA data types. Data types not listed in the following table don't have type-declaration characters.

Data Type	Type-Declaration Character
Integer	%
Long	&

Single	!
Double	#
Currency	@
String	\$

This method of data typing is essentially a holdover from BASIC; it's better to declare your variables by using the other techniques described in this chapter. I list these type declaration characters here just in case you encounter them in an older program.

In the following example, the Dim statement is the first instruction in the module. Both Procedure1 and Procedure2 have access to the CurrentValue variable.

```
Dim CurrentValue as Long
```

```
Sub Procedure1()
```

```
' - [Code goes here] -
```

```
End Sub
```

```
Sub Procedure2()
```

```
' - [Code goes here] -
```

```
End Sub
```

The value of a module-wide variable retains its value when a procedure ends normally (that is, when it reaches the End Sub or End Function statement). An exception is if the procedure is halted with an End statement. When VBA encounters an End statement, all module-wide variables in all modules lose their values.

### **Public variables**

To make a variable available to all the procedures in all the VBA modules in a project, declare the variable at the module level (before the first procedure declaration) by using the Public keyword rather than Dim. Here's an example:

```
Public CurrentRate as Long
```

The Public keyword makes the CurrentRate variable available to any procedure in the VBA project, even those in other modules within the project. You must insert this statement before the first procedure in a module (any module). This type of declaration must appear in a standard VBA module, not in a code module for a sheet or a UserForm.

### **Static variables**

Static variables are a special case. They're declared at the procedure level, and they retain their value when the procedure ends normally. However, if the procedure is halted by an End statement, static variables do lose their values.

You declare static variables by using the Static keyword:

```
Sub MySub()
```

```
Static Counter as Long
```

```
' - [Code goes here] -
```

```
End Sub
```

**Working with constants**

A variable's value may change while a procedure is executing (that's why it's called a variable). Sometimes, you need to refer to a named value or string that never changes: a constant.

Using constants throughout your code in place of hard-coded values or strings is an excellent programming practice. For example, if your procedure needs to refer to a specific value (such as an interest rate) several times, it's better to declare the value as a constant and use the constant's name rather than its value in your expressions. This technique not only makes your code more readable, it also makes it easier to change should the need arise — you have to change only one instruction rather than several.

**Declaring constants**

You declare constants with the Const statement. Here are some examples:

```
Const NumQuarters as Integer = 4
```

```
Const Rate = .0725, Period = 12
```

```
Const ModName as String = "Budget Macros"
```

```
Public Const AppName as String = "Budget Application"
```

**Variable naming conventions**

Some programmers name variables so that users can identify their data types by just looking at their names. Personally, I don't use this technique very often because I think it makes the code more difficult to read, but you might find it helpful.

The naming convention involves using a standard lowercase prefix for the variable's name. For example, if you have a Boolean variable that tracks whether a workbook has been saved, you might name the variable `bWasSaved`. That way, it's clear that the variable is a Boolean variable. The following table lists some standard prefixes for data types:

Data Type	Prefix
Boolean	b
Integer	i
Long	l
Single	s
Double	d
Currency	c
Date/Time	dt
String	str
Object	obj
Variant	v
User-defined	u

The second example doesn't declare a data type. Consequently, VBA determines the data type from the value. The `Rate` variable is a Double, and the

Period variable is an Integer. Because a constant never changes its value, you normally want to declare your constants as a specific data type.

Like variables, constants also have a scope. If you want a constant to be available within a single procedure only, declare it after the Sub or Function statement to make it a local constant. To make a constant available to all procedures in a module, declare it before the first procedure in the module. To make a constant available to all modules in the workbook, use the Public keyword and declare the constant before the first procedure in a module. For example:

```
Public Const InterestRate As Double = 0.0725
```

**Notes:** If your VBA code attempts to change the value of a constant, you get an error (Assignment to constant not permitted). This message is what you would expect. A constant is a constant, not a variable.

### Using predefined constants

Excel and VBA make available many predefined constants, which you can use without declaring. In fact, you don't even need to know the value of these constants to use them. The macro recorder generally uses constants rather than actual values. The following procedure uses a built-in constant (xlLandscape) to set the page orientation to landscape for the active sheet:

```
Sub SetToLandscape()  
    ActiveSheet.PageSetup.Orientation = xlLandscape  
End Sub
```

I discovered the xlLandscape constant by recording a macro. I also could have found this information in the Help system. And, if you have the AutoList Members option turned on, you can often get some assistance while you enter your code (see Figure 8-2). In many cases, VBA lists all the constants that you can assign to a property.

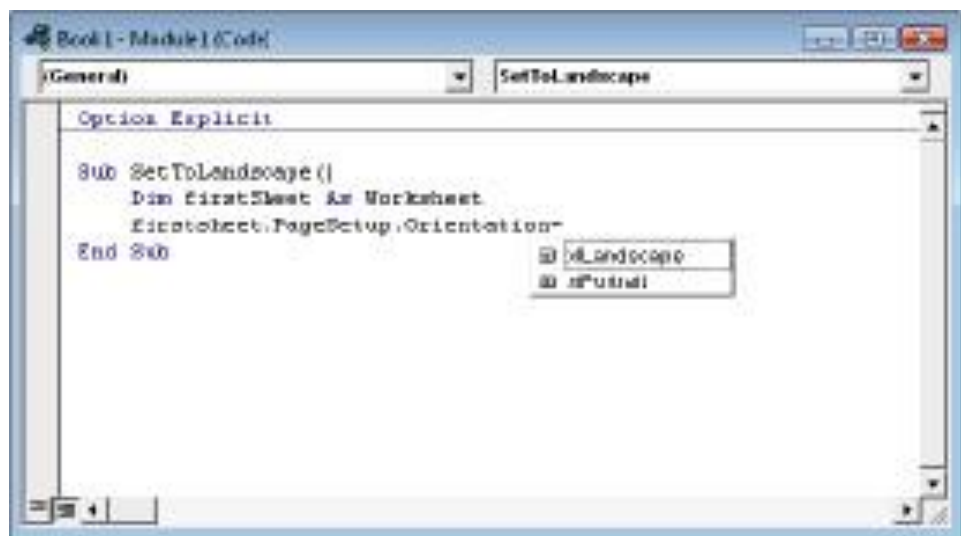


FIGURE 8-2: VBA displays a list of constants that you can assign to a property.

The actual value for xlLandscape is 2 (which you can discover by using the Immediate window). The other built-in constant for changing paper orientation is xlPortrait, which has a value of 1. Obviously, if you use the built-in constants, you don't really need to know their values.



### Working with strings

Like Excel, VBA can manipulate both numbers and text (strings). There are two types of strings in VBA:

- Fixed-length strings are declared with a specified number of characters. The maximum length is 65,535 characters.
- Variable-length strings theoretically can hold up to 2 billion characters.

Each character in a string requires 1 byte of storage, plus a small amount of storage for the header of each string. When you declare a variable with a Dim statement as data type String, you can specify the length if you know it (that is, a fixed-length string), or you can let VBA handle it dynamically (a variable-length string).

In the following example, the MyString variable is declared to be a string with a maximum length of 50 characters. YourString is also declared as a string; but it's a variable-length string, so its length is unfixed.

```
Dim MyString As String * 50
```

```
Dim YourString As String
```

### Working with dates

You can use a string variable to store a date, but if you do, it's not a real date (meaning you can't perform date calculations with it). Using the Date data type is a better way to work with dates.

A variable defined as a date uses 8 bytes of storage and can hold dates ranging from January 1, 0100, to December 31, 9999. That's a span of nearly 10,000 years — more than enough for even the most aggressive financial forecast! The Date data type is also useful for storing time-related data. In VBA, you specify dates and times by enclosing them between two hash marks (#).

**Notes: The range of dates that VBA can handle is much larger than Excel's own date range, which begins with January 1, 1900, and extends through December 31, 1999. Therefore, be careful that you don't attempt to use a date in a worksheet that is outside of Excel's acceptable date range.**

**Notes: In Chapter 10, I describe some relatively simple VBA functions that enable you to create formulas that work with pre-1900 dates in a worksheet.**

### About Excel's date bug

It is commonly known that Excel has a date bug: It incorrectly assumes that the year 1900 is a leap year. Even though there was no February 29, 1900, Excel accepts the following formula and displays the result as the 29th day of February, 1900:

```
=Date(1900,2,29)
```

VBA doesn't have this date bug. The VBA equivalent of Excel's DATE function is DateSerial. The following expression (correctly) returns March 1, 1900:

```
DateSerial(1900,2,29)
```

Therefore, Excel's date serial number system doesn't correspond exactly to the VBA date serial number system. These two systems return different values for dates between January 1, 1900, and February 28, 1900.

Here are some examples of declaring variables and constants as Date data types:

```
Dim Today As Date
```

```
Dim StartTime As Date
```

```
Const FirstDay As Date = #1/1/2010#
```

```
Const Noon = #12:00:00#
```

**Notes: Dates are always defined using month/day/year format, even if your system is set up to display dates in a different format (for example, day/month/year).**

If you use a message box to display a date, it's displayed according to your system's short date format. Similarly, a time is displayed according to your system's time format (either 12- or 24-hour). You can modify these system settings by using the Regional Settings option in the Windows Control Panel.

## 2.4. Assignment Statements

An assignment statement is a VBA instruction that makes a mathematical evaluation and assigns the result to a variable or an object. Excel's Help system defines expression as "a combination of keywords, operators, variables, and constants that yields a string, number, or object. An expression can perform a calculation, manipulate characters, or test data."

I couldn't have said it better myself. Much of the work done in VBA involves developing (and debugging) expressions. If you know how to create formulas in Excel, you'll have no trouble creating expressions in VBA. With a worksheet formula, Excel displays the result in a cell. The result of a VBA expression, on the other hand, can be assigned to a variable or used as a property value.

VBA uses the equal sign (=) as its assignment operator. The following are examples of assignment statements (the expressions are to the right of the equal sign):

```
x = 1
```

```
x = x + 1
```

```
x = (y * 2) / (z * 2)
```

```
FileOpen = True
```

```
FileOpen = Not FileOpen
```

```
Range("TheYear").Value = 2010
```

**Notes: Expressions can be very complex. You may want to use the line continuation sequence (space followed by an underscore) to make lengthy expressions easier to read.**

Often, expressions use functions. These functions can be built-in VBA functions, Excel's worksheet functions, or custom functions that you develop in VBA. I discuss built-in VBA functions later in this chapter (see the upcoming section "Built-in Functions").

Operators play a major role in VBA. Familiar operators describe mathematical operations, including addition (+), multiplication (\*), division (/), subtraction (-), exponentiation (^), and string concatenation (&). Less familiar operators are the backslash (\) (used in integer division) and the Mod operator (used in modulo arithmetic). The Mod operator returns the remainder of one number divided by another. For example, the following expression returns 2:

17 Mod 3

VBA also supports the same comparison operators used in Excel formulas: equal to (=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), and not equal to (<>).

With one exception, the order of precedence for operators in VBA is exactly the same as in Excel (see Table 8-3). And, of course, you can use parentheses to change the natural order of precedence.

**Notes: The negation operator (a minus sign) is handled differently in VBA. In Excel, the following formula returns 25:**

$$=-5^2$$

**In VBA, x equals -25 after this statement is executed:**

$$x = -5 ^ 2$$

**VBA performs the exponentiation operation first and then applies the negation operator. The following statement returns 25:**

$$x = (-5) ^ 2$$

**Table 8-3: Operator Precedence**

Operator	Operation	Order of Precedence
^	Exponentiation	1
* and /	Multiplication and division	2
+ and -	Addition and subtraction	3
&	Concatenation	4
=, <, >, <=, >=, <>	Comparison	5

In the statement that follows, x is assigned the value 10 because the multiplication operator has a higher precedence than the addition operator.

$$x = 4 + 3 * 2$$

To avoid ambiguity, you may prefer to write the statement as follows:

$$x = 4 + (3 * 2)$$

In addition, VBA provides a full set of logical operators, shown in Table 8-4. For complete details on these operators (including examples), use the VBA Help system.

**Table 8-4: VBA Logical Operators**

Operator	What It Does
Not	Performs a logical negation on an expression.
And	Performs a logical conjunction on two expressions.
Or	Performs a logical disjunction on two expressions.
Xor	Performs a logical exclusion on two expressions.

Eqv	Performs a logical equivalence on two expressions.
Imp	Performs a logical implication on two expressions.

The following instruction uses the Not operator to toggle the gridline display in the active window. The DisplayGridlines property takes a value of either True or False. Therefore, using the Not operator changes False to True and True to False.

```
ActiveWindow.DisplayGridlines = _
Not ActiveWindow.DisplayGridlines
```

The following expression performs a logical And operation. The MsgBox statement displays True only when Sheet1 is the active sheet and the active cell is in Row 1. If either or both of these conditions aren't true, the MsgBox statement displays False.

```
MsgBox ActiveSheet.Name = "Sheet1" And ActiveCell.Row = 1
```

The following expression performs a logical Or operation. The MsgBox statement displays True when either Sheet1 or Sheet2 is the active sheet.

```
MsgBox ActiveSheet.Name = "Sheet1" Or ActiveSheet.Name = "Sheet2"
```

## 2.5. Arrays

An array is a group of elements of the same type that have a common name. You refer to a specific element in the array by using the array name and an index number. For example, you can define an array of 12 string variables so that each variable corresponds to the name of a month. If you name the array MonthNames, you can refer to the first element of the array as MonthNames(0), the second element as MonthNames(1), and so on, up to MonthNames(11).

### Declaring arrays

You declare an array with a Dim or Public statement, just as you declare a regular variable. You can also specify the number of elements in the array. You do so by specifying the first index number, the keyword To, and the last index number — all inside parentheses. For example, here's how to declare an array comprising exactly 100 integers:

```
Dim MyArray(1 To 100) As Integer
```

**Notes: When you declare an array, you need specify only the upper index, in which case VBA assumes that 0 is the lower index. Therefore, the two statements that follow have the same effect:**

```
Dim MyArray(0 to 100) As Integer
```

```
Dim MyArray(100) As Integer
```

**In both cases, the array consists of 101 elements.**

By default, VBA assumes zero-based arrays. If you would like VBA to assume that 1 is the lower index for all arrays that declare only the upper index, include the following statement before any procedures in your module:

```
Option Base 1
```

### Declaring multidimensional arrays

The array examples in the preceding section are one-dimensional arrays. VBA arrays can have up to 60 dimensions, although you'll rarely need more than

three dimensions (a 3-D array). The following statement declares a 100-integer array with two dimensions (2-D):

```
Dim MyArray(1 To 10, 1 To 10) As Integer
```

You can think of the preceding array as occupying a 10-x-10 matrix. To refer to a specific element in a 2-D array, you need to specify two index numbers. For example, here's how you can assign a value to an element in the preceding array:

```
MyArray(3, 4) = 125
```

Following is a declaration for a 3-D array that contains 1,000 elements (visualize this array as a cube):

```
Dim MyArray(1 To 10, 1 To 10, 1 To 10) As Integer
```

Reference an item within the array by supplying three index numbers:

```
MyArray(4, 8, 2) = 0
```

### Declaring dynamic arrays

A dynamic array doesn't have a preset number of elements. You declare a dynamic array with a blank set of parentheses:

```
Dim MyArray() As Integer
```

Before you can use a dynamic array in your code, however, you must use the ReDim statement to tell VBA how many elements are in the array. You can use a variable to assign the number of elements in an array. Often the value of the variable isn't known until the procedure is executing. For example, if the variable x contains a number, you can define the array's size by using this statement:

```
ReDim MyArray (1 to x)
```

You can use the ReDim statement any number of times, changing the array's size as often as you need to. When you change an array's dimensions the existing values are destroyed. If you want to preserve the existing values, use ReDim Preserve. For example:

```
ReDim Preserve MyArray (1 to y)
```

Arrays crop up later in this chapter when I discuss looping ("Looping blocks of instructions").

## 2.6. Object Variables

An object variable is a variable that represents an entire object, such as a range or a worksheet. Object variables are important for two reasons:

- They can simplify your code significantly.
- They can make your code execute more quickly.

Object variables, like normal variables, are declared with the Dim or Public statement. For example, the following statement declares InputArea as a Range object variable:

```
Dim InputArea As Range
```

Use the Set keyword to assign an object to the variable. For example:

```
Set InputArea = Range("C16:E16")
```

To see how object variables simplify your code, examine the following procedure, which doesn't use an object variable:

```
Sub NoObjVar()
```

```
Worksheets("Sheet1").Range("A1").Value = 124
```

```
Worksheets("Sheet1").Range("A1").Font.Bold = True
Worksheets("Sheet1").Range("A1").Font.Italic = True
Worksheets("Sheet1").Range("A1").Font.Size = 14
Worksheets("Sheet1").Range("A1").Font.Name = "Cambria"
End Sub
```

This routine enters a value into cell A1 of Sheet1 on the active workbook, applies some formatting, and changes the fonts and size. That's a lot of typing. To reduce wear and tear on your fingers (and make your code more efficient), you can condense the routine with an object variable:

```
Sub ObjVar()
Dim MyCell As Range
Set MyCell = Worksheets("Sheet1").Range("A1")
MyCell.Value = 124
MyCell.Font.Bold = True
MyCell.Font.Italic = True
MyCell.Font.Size = 14
MyCell.Font.Name = Cambria
End Sub
```

After the variable MyCell is declared as a Range object, the Set statement assigns an object to it. Subsequent statements can then use the simpler MyCell reference in place of the lengthy Worksheets("Sheet1").Range("A1") reference.

**Notes:** After an object is assigned to a variable, VBA can access it more quickly than it can a normal, lengthy reference that has to be resolved. So when speed is critical, use object variables. One way to think about code efficiency is in terms of dot processing. Every time VBA encounters a dot, as in Sheets(1).Range("A1"), it takes time to resolve the reference. Using an object variable reduces the number of dots to be processed. The fewer the dots, the faster the processing time. Another way to improve the speed of your code is by using the With-End With construct, which also reduces the number of dots to be processed. I discuss this construct later in this chapter.

## 2.7. User-Defined Data Types

VBA lets you create custom, or user-defined, data types. A user-defined data type can ease your work with some types of data. For example, if your application deals with customer information, you may want to create a user-defined data type named CustomerInfo:

- Type CustomerInfo
- Company As String
- Contact As String
- RegionCode As Long
- Sales As Double
- End Type

**Notes:** You define custom data types at the top of your module, before any procedures.

After you create a user-defined data type, you use a Dim statement to declare a variable as that type. Usually, you define an array. For example:

```
Dim Customers(1 To 100) As CustomerInfo
```

Each of the 100 elements in this array consists of four components (as specified by the user-defined data type, CustomerInfo). You can refer to a particular component of the record as follows:

```
Customers(1).Company = "Acme Tools"
```

```
Customers(1).Contact = "Tim Robertson"
```

```
Customers(1).RegionCode = 3
```

```
Customers(1).Sales = 150674.98
```

You can also work with an element in the array as a whole. For example, to copy the information from Customers(1) to Customers(2), use this instruction:

```
Customers(2) = Customers(1)
```

The preceding example is equivalent to the following instruction block:

```
Customers(2).Company = Customers(1).Company
```

```
Customers(2).Contact = Customers(1).Contact
```

```
Customers(2).RegionCode = Customers(1).RegionCode
```

```
Customers(2).Sales = Customers(1).Sales
```

## 2.8. Built-in Functions

Like most programming languages, VBA has a variety of built-in functions that simplify calculations and operations. Many VBA functions are similar (or identical) to Excel worksheet functions. For example, the VBA function UCase, which converts a string argument to uppercase, is equivalent to the Excel worksheet function UPPER.

**Notes:** Appendix B contains a complete list of VBA functions, with a brief description of each. All are thoroughly described in the VBA Help system.

**Notes:** To get a list of VBA functions while you're writing your code, type VBA followed by a period (.). The VBE displays a list of all its members, including functions (see Figure 8-3). The functions are preceded by a green icon.

If this technique doesn't work for you, make sure that the Auto List Members option is selected. Choose Tools⇒Options and then click the Editor tab.

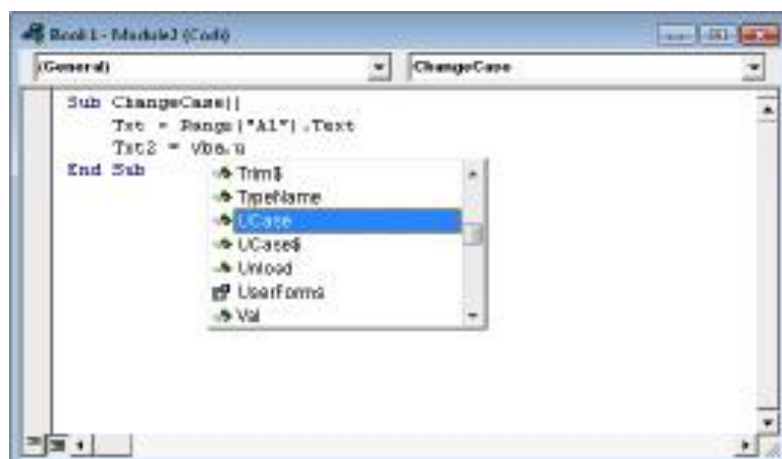


FIGURE 8-3: Displaying a list of VBA functions in the VBE.

You use functions in VBA expressions in much the same way that you use functions in worksheet formulas. Here's a simple procedure that calculates the square root of a variable (using the VBA Sqr function), stores the result in another variable, and then displays the result:

```
Sub ShowRoot()  
    Dim MyValue As Double  
    Dim SquareRoot As Double  
    MyValue = 25  
    SquareRoot = Sqr(MyValue)  
    MsgBox SquareRoot  
End Sub
```

The VBA Sqr function is equivalent to the Excel SQRT worksheet function.

You can use many (but not all) of Excel's worksheet functions in your VBA code. The WorksheetFunction object, which is contained in the Application object, holds all the worksheet functions that you can call from your VBA procedures.

To use a worksheet function in a VBA statement, just precede the function name with

*Application.WorksheetFunction*

### The MsgBox function

The MsgBox function is one of the most useful VBA functions. Many of the examples in this chapter use this function to display the value of a variable.

This function often is a good substitute for a simple custom dialog box. It's also an excellent debugging tool because you can insert MsgBox functions at any time to pause your code and display the result of a calculation or assignment.

Most functions return a single value, which you assign to a variable. The MsgBox function not only returns a value but also displays a dialog box that the user can respond to. The value returned by the MsgBox function represents the user's response to the dialog box. You can use the MsgBox function even when you have no interest in the user's response but want to take advantage of the message display.

The official syntax of the MsgBox function has five arguments (those in square brackets are optional):

MsgBox(prompt[, buttons][, title][, helpfile, context])

- **prompt:** (Required) The message displayed in the pop-up display.
- **buttons:** (Optional) A value that specifies which buttons and which icons, if any, appear in the message box. Use built-in constants — for example, vbYesNo.
- **title:** (Optional) The text that appears in the message box's title bar. The default is Microsoft Excel.
- **helpfile:** (Optional) The name of the Help file associated with the message box.
- **context:** (Optional) The context ID of the Help topic. This represents a specific Help topic to display. If you use the context argument, you must also use the helpfile argument.

You can assign the value returned to a variable, or you can use the function by itself without an assignment statement. This example assigns the result to the variable Ans:



```
Ans = MsgBox("Continue?", vbYesNo + vbQuestion, "Tell me")
```

```
If Ans = vbNo Then Exit Sub
```

Notice that I used the sum of two built-in constants (vbYesNo + vbQuestion) for the buttons argument. Using vbYesNo displays two buttons in the message box: one labeled Yes and one labeled No. Adding vbQuestion to the argument also displays a question mark icon. When the first statement is executed, Ans contains one of two values, represented by the constant vbYes or vbNo. In this example, if the user clicks the No button, the procedure ends.

The following example demonstrates how to use an Excel worksheet function in a VBA procedure. Excel's infrequently used ROMAN function converts a decimal number into a Roman numeral.

```
Sub ShowRoman()
```

```
Dim DecValue As Long
```

```
Dim RomanValue As String
```

```
DecValue = 1939
```

```
RomanValue = Application.WorksheetFunction.Roman(DecValue)
```

```
MsgBox RomanValue
```

```
End Sub
```

When you execute this procedure, the MsgBox function displays the string MCMXXXIX. Fans of old movies are often dismayed when they learn that Excel doesn't have a function to convert a Roman numeral to its decimal equivalent.

Keep in mind that you can't use worksheet functions that have an equivalent VBA function. For example, VBA can't access the Excel SQRT worksheet function because VBA has its own version of that function: Sqr. Therefore, the following statement generates an error:

```
MsgBox Application.WorksheetFunction.Sqrt(123) 'error
```

**Notes:** As I describe in Chapter 10, you can use VBA to create custom worksheet functions that work just like Excel's built-in worksheet functions.

## 2.9. Manipulating Objects and Collections

As an Excel programmer, you'll spend a lot of time working with objects and collections. Therefore, you want to know the most efficient ways to write your code to manipulate these objects and collections. VBA offers two important constructs that can simplify working with objects and collections:

- *With-End With constructs*
- *For Each-Next constructs*

### With-End With constructs

The With-End With construct enables you to perform multiple operations on a single object. To start understanding how the With-End With construct works, examine the following procedure, which modifies six properties of a selection's formatting (the selection is assumed to be a Range object):

```
Sub ChangeFont1()
```

```
Selection.Font.Name = "Cambria"
```

```
Selection.Font.Bold = True
```

```
Selection.Font.Italic = True  
Selection.Font.Size = 12  
Selection.Font.Underline = xlUnderlineStyleSingle  
Selection.Font.ThemeColor = xlThemeColorAccent1  
End Sub
```

You can rewrite this procedure using the With-End With construct. The following procedure performs exactly like the preceding one:

```
Sub ChangeFont2()  
With Selection.Font  
.Name = "Cambria"  
.Bold = True  
.Italic = True  
.Size = 12  
.Underline = xlUnderlineStyleSingle  
.ThemeColor = xlThemeColorAccent1  
End With  
End Sub
```

Some people think that the second incarnation of the procedure is actually more difficult to read. Remember, though, that the objective is increased speed. Although the first version may be more straightforward and easier to understand, a procedure that uses the With-End With construct to change several properties of an object can be faster than the equivalent procedure that explicitly references the object in each statement.

**Notes:** When you record a VBA macro, Excel uses the With-End With construct every chance it gets. To see a good example of this construct, try recording your actions while you change the page orientation using the Page Layout⇒Page Setup⇒Orientation command.

### For Each-Next constructs

Recall from the preceding chapter that a collection is a group of related objects. For example, the Workbooks collection is a collection of all open Workbook objects, and there are many other collections that you can work with.

Suppose that you want to perform some action on all objects in a collection. Or suppose that you want to evaluate all objects in a collection and take action under certain conditions. These occasions are perfect for the For Each-Next construct because you don't have to know how many elements are in a collection to use the For Each-Next construct.

The syntax of the For Each-Next construct is

```
For Each element In collection  
[instructions]  
[Exit For]  
[instructions]  
Next [element]
```

The following procedure uses the For Each-Next construct with the Worksheets collection in the active workbook. When you execute the procedure, the MsgBox

function displays each worksheet's Name property. (If five worksheets are in the active workbook, the MsgBox function is called five times.)

```
Sub CountSheets()  
Dim Item as Worksheet  
For Each Item In ActiveWorkbook.Worksheets  
MsgBox Item.Name  
Next Item  
End Sub
```

**Notes:** In the preceding example, Item is an object variable (more specifically, a Worksheet object). There's nothing special about the name Item; you can use any valid variable name in its place.

The next example uses For Each-Next to cycle through all objects in the Windows collection and count the number of windows that are hidden.

```
Sub HiddenWindows()  
Dim Cnt As Integer  
Dim Win As Window  
Cnt = 0  
For Each Win In Windows  
If Not Win.Visible Then Cnt = Cnt + 1  
Next Win  
MsgBox Cnt & " hidden windows."  
End Sub
```

For each window, if the window is hidden, the Cnt variable is incremented. When the loop ends, the message box displays the value of Cnt.

Here's an example that closes all workbooks except the active workbook. This procedure uses the If-Then construct to evaluate each workbook in the Workbooks collection.

```
Sub CloseInactive()  
Dim Book as Workbook  
For Each Book In Workbooks  
If Book.Name <> ActiveWorkbook.Name Then Book.Close  
Next Book  
End Sub
```

A common use for the For Each-Next construct is to loop through all cells in a range. The next example of For Each-Next is designed to be executed after the user selects a range of cells. Here, the Selection object acts as a collection that consists of Range objects because each cell in the selection is a Range object. The procedure evaluates each cell and uses the VBA UCase function to convert its contents to uppercase. (Numeric cells are not affected.)

```
Sub MakeUpperCase()  
Dim Cell as Range  
For Each Cell In Selection  
Cell.Value = UCase(Cell.Value)
```

## 2.10. Controlling Code Execution

*Next Cell*

*End Sub*

VBA provides a way to exit a For-Next loop before all the elements in the collection are evaluated. Do this with an Exit For statement. The example that follows selects the first negative value in Row 1 of the active sheet:

*Sub SelectNegative()*

*Dim Cell As Range*

*For Each Cell In Range("1:1")*

*If Cell.Value < 0 Then*

*Cell.Select*

*Exit For*

*End If*

*Next Cell*

*End Sub*

This example uses an If-Then construct to check the value of each cell. If a cell is negative, it's selected, and then the loop ends when the Exit For statement is executed.

Some VBA procedures start at the top and progress line by line to the bottom. Macros that you record, for example, always work in this fashion. Often, however, you need to control the flow of your routines by skipping over some statements, executing some statements multiple times, and testing conditions to determine what the routine does next.

The preceding section describes the For Each-Next construct, which is a type of loop. This section discusses the additional ways of controlling the execution of your VBA procedures:

- GoTo statements
- If-Then constructs
- Select Case constructs
- For-Next loops
- Do While loops
- Do Until loops

### GoTo statements

The most straightforward way to change the flow of a program is to use a GoTo statement. This statement simply transfers program execution to a new instruction, which must be preceded by a label (a text string followed by a colon, or a number with no colon). VBA procedures can contain any number of labels, but a GoTo statement can't branch outside of a procedure.

The following procedure uses the VBA InputBox function to get the user's name. If the name is not Howard, the procedure branches to the WrongName label and ends. Otherwise, the procedure executes some additional code. The Exit Sub statement causes the procedure to end.

*Sub GoToDemo()*

*UserName = InputBox("Enter Your Name:")*

*If UserName <> "Howard" Then GoTo WrongName*

```
MsgBox ("Welcome Howard...")  
'-[More code here] -  
Exit Sub  
  
WrongName:  
MsgBox "Sorry. Only Howard can run this macro."  
End Sub
```

This simple procedure works, but it's not an example of good programming. In general, you should use the GoTo statement only when you have no other way to perform an action. In fact, the only time you really need to use a GoTo statement in VBA is for error handling (refer to Chapter 9).

Finally, it goes without saying that the preceding example is not intended to demonstrate an effective security technique!

### If-Then constructs

Perhaps the most commonly used instruction grouping in VBA is the If-Then construct. This common instruction is one way to endow your applications with decision-making capability. Good decision-making is the key to writing successful programs.

The basic syntax of the If-Then construct is

```
If condition Then true_instructions [Else false_instructions]
```

The If-Then construct is used to execute one or more statements conditionally. The Else clause is optional. If included, the Else clause lets you execute one or more instructions when the condition that you're testing isn't True.

The following procedure demonstrates an If-Then structure without an Else clause. The example deals with time, and VBA uses a date-and-time serial number system similar to Excel's. The time of day is expressed as a fractional value — for example, noon is represented as .5. The VBA Time function returns a value that represents the time of day, as reported by the system clock. In the following example, a message is displayed if the time is before noon. If the current system time is greater than or equal to .5, the procedure ends, and nothing happens.

```
Sub GreetMe1()  
If Time < 0.5 Then MsgBox "Good Morning"  
End Sub
```

Another way to code this routine is to use multiple statements, as follows:

```
Sub GreetMe1a()  
If Time < 0.5 Then  
MsgBox "Good Morning"  
End If  
End Sub
```

Notice that the If statement has a corresponding End If statement. In this example, only one statement is executed if the condition is True. You can, however, place any number of statements between the If and End If statements.

If you want to display a different greeting when the time of day is after noon, add another If-Then statement, like so:

```
Sub GreetMe2()  
If Time < 0.5 Then MsgBox "Good Morning"
```

```
If Time >= 0.5 Then MsgBox "Good Afternoon"
```

```
End Sub
```

Notice that I used >= (greater than or equal to) for the second If-Then statement. This covers the remote chance that the time is precisely 12:00 noon.

Another approach is to use the Else clause of the If-Then construct. For example,

```
Sub GreetMe3()
```

```
If Time < 0.5 Then MsgBox "Good Morning" Else _
```

```
MsgBox "Good Afternoon"
```

```
End Sub
```

Notice that I used the line continuation sequence; If-Then-Else is actually a single statement.

If you need to execute multiple statements based on the condition, use this form:

```
Sub GreetMe3a()
```

```
If Time < 0.5 Then
```

```
MsgBox "Good Morning"
```

```
' Other statements go here
```

```
Else
```

```
MsgBox "Good Afternoon"
```

```
' Other statements go here
```

```
End If
```

```
End Sub
```

If you need to expand a routine to handle three conditions (for example, morning, afternoon, and evening), you can use either three If-Then statements or a form that uses ElseIf. The first approach is simpler:

```
Sub GreetMe4()
```

```
If Time < 0.5 Then MsgBox "Good Morning"
```

```
If Time >= 0.5 And Time < 0.75 Then MsgBox "Good Afternoon"
```

```
If Time >= 0.75 Then MsgBox "Good Evening"
```

```
End Sub
```

The value 0.75 represents 6:00 p.m. — three-quarters of the way through the day and a good point at which to call it an evening.

In the preceding examples, every instruction in the procedure gets executed, even if the first condition is satisfied (that is, it's morning). A more efficient procedure would include a structure that ends the routine when a condition is found to be True. For example, it might display the Good Morning message in the morning and then exit without evaluating the other, superfluous conditions. True, the difference in speed is inconsequential when you design a procedure as small as this routine. But for more complex applications, you need another syntax:

```
If condition Then
```

```
[true_instructions]
```

```
[Elseif condition-n Then
```

```
[alternate_instructions]]
```

```
[Else
```

```
[default_instructions]]
```

```
End If
```

Here's how you can use this syntax to rewrite the GreetMe procedure:

```
Sub GreetMe5()
```

```
If Time < 0.5 Then
```

```
MsgBox "Good Morning"
```

```
Elseif Time >= 0.5 And Time < 0.75 Then
```

```
MsgBox "Good Afternoon"
```

```
Else
```

```
MsgBox "Good Evening"
```

```
End If
```

```
End Sub
```

With this syntax, when a condition is True, the conditional statements are executed, and the If-Then construct ends. In other words, the extraneous conditions aren't evaluated. Although this syntax makes for greater efficiency, some find the code to be more difficult to understand.

The following procedure demonstrates yet another way to code this example. It uses nested If-Then-Else constructs (without using Elseif). This procedure is efficient and also easy to understand. Note that each If statement has a corresponding End If statement.

```
Sub GreetMe6()
```

```
If Time < 0.5 Then
```

```
MsgBox "Good Morning"
```

```
Else
```

```
If Time >= 0.5 And Time < 0.75 Then
```

```
MsgBox "Good Afternoon"
```

```
Else
```

```
If Time >= 0.75 Then
```

```
MsgBox "Good Evening"
```

```
End If
```

```
End If
```

```
End If
```

```
End Sub
```

The following is another example that uses the simple form of the If-Then construct. This procedure prompts the user for a value for Quantity and then displays the appropriate discount based on that value. Note that Quantity is declared as a Variant data type. This is because Quantity contains an empty string (not a numeric value) if the InputBox is cancelled. To keep it simple, this procedure doesn't perform any other error checking. For example, it doesn't ensure that the quantity entered is a non-negative numeric value.

```
Sub Discount1()
```

```
Dim Quantity As Variant
```

```
Dim Discount As Double
```

```
Quantity = InputBox("Enter Quantity: ")
```

```
If Quantity = "" Then Exit Sub
If Quantity >= 0 Then Discount = 0.1
If Quantity >= 25 Then Discount = 0.15
If Quantity >= 50 Then Discount = 0.2
If Quantity >= 75 Then Discount = 0.25
MsgBox "Discount: " & Discount
End Sub
```

Notice that each If-Then statement in this procedure is always executed, and the value for Discount can change. The final value, however, is the desired value.

The following procedure is the previous one rewritten to use the alternate syntax. In this case, the procedure ends after executing the True instruction block.

```
Sub Discount2()
Dim Quantity As Variant
Dim Discount As Double
Quantity = InputBox("Enter Quantity: ")
If Quantity = "" Then Exit Sub
If Quantity >= 0 And Quantity < 25 Then
Discount = 0.1
ElseIf Quantity < 50 Then
Discount = 0.15
ElseIf Quantity < 75 Then
Discount = 0.2
Else
Discount = 0.25
End If
MsgBox "Discount: " & Discount
End Sub
```

I find nested If-Then structures rather cumbersome. As a result, I usually use the If-Then structure only for simple binary decisions. When you need to choose among three or more alternatives, the Select Case structure (discussed next) is often a better construct to use.

### VBA's If function

VBA offers an alternative to the If-Then construct: the If function. This function takes three arguments and works much like Excel's IF worksheet function. The syntax is

If(expr, truepart, falsepart)

- expr: (Required) Expression you want to evaluate.
- truepart: (Required) Value or expression returned if expr is True.
- falsepart: (Required) Value or expression returned if expr is False.

The following instruction demonstrates the use of the If function. The message box displays Zero if cell A1 contains a zero or is empty and displays Nonzero if cell A1 contains anything else.



```
MsgBox If(Range("A1") = 0, "Zero", "Nonzero")
```

It's important to understand that the third argument (falsepart) is always evaluated, even if the first argument (expr) is True. Therefore, the following statement generates a Division By Zero error if the value of n is 0 (zero):

```
MsgBox If(n = 0, 0, 1 / n)
```

### Select Case constructs

The Select Case construct is useful for choosing among three or more options. this construct also works with two options and is a good alternative to If-Then-Else. The syntax for Select Case is as follows:

```
Select Case testexpression  
[Case expressionlist-n  
[instructions-n]]  
[Case Else  
[default_instructions]]  
End Select
```

The following example of a Select Case construct shows another way to code the GreetMe examples that I presented in the preceding section:

```
Sub GreetMe()  
Dim Msg As String  
Select Case Time  
Case Is < 0.5  
Msg = "Good Morning"  
Case 0.5 To 0.75  
Msg = "Good Afternoon"  
Case Else  
Msg = "Good Evening"  
End Select  
MsgBox Msg  
End Sub
```

And here's a rewritten version of the Discount example using a Select Case construct. This procedure assumes that Quantity is always an integer value. For simplicity, the procedure performs no error checking.

```
Sub Discount3()  
Dim Quantity As Variant  
Dim Discount As Double  
Quantity = InputBox("Enter Quantity: ")  
Select Case Quantity  
Case ""  
Exit Sub  
Case 0 To 24  
Discount = 0.1  
Case 25 To 49
```

```
Discount = 0.15
Case 50 To 74
Discount = 0.2
Case Is >= 75
Discount = 0.25
End Select
MsgBox "Discount: " & Discount
End Sub
```

The Case statement also can use a comma to separate multiple values for a single case. The following procedure uses the VBA WeekDay function to determine whether the current day is a weekend (that is, the WeekDay function returns 1 or 7). The procedure then displays an appropriate message.

```
Sub GreetUser1()
Select Case Weekday(Now)
Case 1, 7
MsgBox "This is the weekend"
Case Else
MsgBox "This is not the weekend"
End Select
End Sub
```

The following example shows another way to code the previous procedure:

```
Sub GreetUser2()
Select Case Weekday(Now)
Case 2, 3, 4, 5, 6
MsgBox "This is not the weekend"
Case Else
MsgBox "This is the weekend"
End Select
End Sub
```

Any number of instructions can be written below each Case statement, and they're all executed if that case evaluates to True. If you use only one instruction per case, as in the preceding example, you might want to put the instruction on the same line as the Case keyword (but don't forget the VBA statement-separator character, the colon). This technique makes the code more compact. For example:

```
Sub Discount3()
Dim Quantity As Variant
Dim Discount As Double
Quantity = InputBox("Enter Quantity: ")
Select Case Quantity
Case "": Exit Sub
Case 0 To 24: Discount = 0.1
Case 25 To 49: Discount = 0.15
```

```
Case 50 To 74: Discount = 0.2
Case Is >= 75: Discount = 0.25
End Select
MsgBox "Discount: " & Discount
End Sub
```

**Notes: VBA exits a Select Case construct as soon as a True case is found. Therefore, for maximum efficiency, you should check the most likely case first.**

Select Case structures can also be nested. The following procedure, for example, uses the VBA TypeName function to determine what is selected (a range, nothing, or anything else). If a range is selected, the procedure executes a nested Select Case and tests for the number of cells in the range. If one cell is selected, it displays One cell is selected. Otherwise, it displays a message with the number of selected rows.

```
Sub SelectionType()
Select Case TypeName(Selection)
Case "Range"
Select Case Selection.Count
Case 1
MsgBox "One cell is selected"
Case Else
MsgBox Selection.Rows.Count & " rows"
End Select
Case "Nothing"
MsgBox "Nothing is selected"
Case Else
MsgBox "Something other than a range"
End Select
End Sub
```

This procedure also demonstrates the use of Case Else, a catch-all case. You can nest Select Case constructs as deeply as you need, but make sure that each Select Case statement has a corresponding End Select statement.

This procedure demonstrates the value of using indentation in your code to clarify the structure. For example, take a look at the same procedure without the indentations:

```
Fairly incomprehensible, eh? Sub SelectionType()
Select Case TypeName(Selection)
Case "Range"
Select Case Selection.Count
Case 1
MsgBox "One cell is selected"
Case Else
MsgBox Selection.Rows.Count & " rows"
```

```
End Select
Case "Nothing"
MsgBox "Nothing is selected"
Case Else
MsgBox "Something other than a range"
End Select
End Sub
```

### Looping blocks of instructions

Looping is the process of repeating a block of instructions. You might know the number of times to loop, or the number may be determined by the values of variables in your program.

The following code, which enters consecutive numbers into a range, demonstrates what I call a bad loop. The procedure uses two variables to store a starting value (StartVal) and the total number of cells to fill (NumToFill). This loop uses the GoTo statement to control the flow. If the Cnt variable, which keeps track of how many cells are filled, is less than the value of NumToFill, the program control loops back to DoAnother.

```
Sub BadLoop()
Dim StartVal As Integer
Dim NumToFill As Integer
Dim Cnt As Integer
StartVal = 1
NumToFill = 100
ActiveCell.Value = StartVal
Cnt = 1
DoAnother:
ActiveCell.Offset(Cnt, 0).Value = StartVal + Cnt
Cnt = Cnt + 1
If Cnt < NumToFill Then GoTo DoAnother Else Exit Sub
End Sub
```

This procedure works as intended, so why is it an example of bad looping? Programmers generally frown on using a GoTo statement when not absolutely necessary. Using GoTo statements to loop is contrary to the concept of structured coding. (See the "What is structured programming?" sidebar.) In fact, a GoTo statement makes the code much more difficult to read because representing a loop using line indentations is almost impossible. In addition, this type of unstructured loop makes the procedure more susceptible to error. Furthermore, using lots of labels results in spaghetti code — code that appears to have little or no structure and flows haphazardly.

Because VBA has several structured looping commands, you almost never have to rely on GoTo statements for your decision-making.

### For-Next loops

The simplest type of a good loop is a For-Next loop. Its syntax is

```
For counter = start To end [Step stepval]
```

*[instructions]*

*[Exit For]*

*[instructions]*

*Next [counter]*

Following is an example of a For-Next loop that doesn't use the optional Step value or the optional Exit For statement. This routine executes the `Sum = Sum + Sqr(Count)` statement 100 times and displays the result — that is, the sum of the square roots of the first 100 integers.

```
Sub SumSquareRoots()  
    Dim Sum As Double  
    Dim Count As Integer  
    Sum = 0  
    For Count = 1 To 100  
        Sum = Sum + Sqr(Count)  
    Next Count  
    MsgBox Sum  
End Sub
```

**Notes:** When you use For-Next loops, it's important to understand that the loop counter is a normal variable — nothing special. As a result, it's possible to change the value of the loop counter within the block of code executed between the For and Next statements. Changing the loop counter inside of a loop, however, is a bad practice and can cause unpredictable results. In fact, you should take precautions to ensure that your code doesn't change the loop counter.

You can also use a Step value to skip some values in the loop. Here's the same procedure rewritten to sum the square roots of the odd numbers between 1 and 100:

```
Sub SumOddSquareRoots()  
    Dim Sum As Double  
    Dim Count As Integer  
    Sum = 0  
    For Count = 1 To 100 Step 2  
        Sum = Sum + Sqr(Count)  
    Next Count  
    MsgBox Sum  
End Sub
```

In this procedure, Count starts out as 1 and then takes on values of 3, 5, 7, and so on. The final value of Count used within the loop is 99. When the loop ends, the value of Count is 101.

A Step value in a For-Next loop can also be negative. The procedure that follows deletes Rows 2, 4, 6, 8, and 10 of the active worksheet:

```
Sub DeleteRows()  
    Dim RowNum As Long  
    For RowNum = 10 To 2 Step -2
```

```
Rows(RowNum).Delete  
Next RowNum  
End Sub
```

You may wonder why I used a negative Step value in the DeleteRows procedure. If you use a positive Step value, as shown in the following procedure, incorrect rows are deleted. That's because the row numbers below a deleted row get a new row number. For example, when Row 2 is deleted, Row 3 becomes the new Row 2. Using a negative Step value ensures that the correct rows are deleted.

```
Sub DeleteRows2()  
Dim RowNum As Long  
For RowNum = 2 To 10 Step 2  
Rows(RowNum).Delete  
Next RowNum  
End Sub
```

The following procedure performs the same task as the BadLoop example found at the beginning of the "Looping blocks of instructions" section. I eliminate the GoTo statement, however, converting a bad loop into a good loop that uses the For-Next structure.

```
Sub GoodLoop()  
Dim StartVal As Integer  
Dim NumToFill As Integer  
Dim Cnt As Integer  
StartVal = 1  
NumToFill = 100  
For Cnt = 0 To NumToFill - 1  
ActiveCell.Offset(Cnt, 0).Value = StartVal + Cnt  
Next Cnt  
End Sub
```

For-Next loops can also include one or more Exit For statements within the loop. When this statement is encountered, the loop terminates immediately and control passes to the statement following the Next statement of the current For-Next loop. The following example demonstrates use of the Exit For statement. This procedure determines which cell has the largest value in Column A of the active worksheet:

```
Sub ExitForDemo()  
Dim MaxVal As Double  
Dim Row As Long  
MaxVal = Application.WorksheetFunction.Max(Range("A:A"))  
For Row = 1 To 1048576  
If Cells(Row, 1).Value = MaxVal Then  
Exit For  
End If  
Next Row  
MsgBox "Max value is in Row " & Row
```

```
Cells(Row, 1).Activate
```

```
End Sub
```

The maximum value in the column is calculated by using the Excel MAX function, and the value is assigned to the MaxVal variable. The For-Next loop checks each cell in the column. If the cell being checked is equal to MaxVal, the Exit For statement terminates the loop and the statements following the Next statement are executed. These statements display the row of the maximum value and activate the cell.

**Notes: The ExitForDemo procedure is presented to demonstrate how to exit from a For-Next loop. However, it's not the most efficient way to activate the largest value in a range. In fact, a single statement does the job:**

```
Range("A:A").Find(Application.WorksheetFunction.Max  
(Range("A:A"))).Activate
```

The previous examples use relatively simple loops. But you can have any number of statements in the loop, and you can even nest For-Next loops inside other For-Next loops. Here's an example that uses nested For-Next loops to initialize a 10 x 10 x 10 array with the value -1. When the procedure is finished, each of the 1,000 elements in MyArray contains -1.

```
Sub NestedLoops()  
Dim MyArray(1 to 10, 1 to 10, 1 to 10)  
Dim i As Integer, j As Integer, k As Integer  
For i = 1 To 10  
For j = 1 To 10  
For k = 1 To 10  
MyArray(i, j, k) = -1  
Next k  
Next j  
Next i  
' [More code goes here]  
End Sub
```

### Do While loops

This section describes another type of looping structure available in VBA. Unlike a For-Next loop, a Do While loop executes as long as a specified condition is met.

A Do While loop can have either of two syntaxes:

```
Do [While condition]  
[instructions]  
[Exit Do]  
[instructions]  
Loop
```

or

```
Do
```

*[instructions]*

*[Exit Do]*

*[instructions]*

*Loop [While condition]*

As you can see, VBA lets you put the While condition at the beginning or the end of the loop. The difference between these two syntaxes involves the point in time when the condition is evaluated. In the first syntax, the contents of the loop may never be executed. In the second syntax, the statements inside the loop are always executed at least one time.

The following examples insert a series of dates into the active worksheet. The dates correspond to the days in the current month, and the dates are entered in a column beginning at the active cell.

**These examples use some VBA date-related functions:**

- Date returns the current date.
- Month returns the month number for a date supplied as its argument.
- DateSerial returns a date for the year, month, and day supplied as arguments.

The first example demonstrates a Do While loop that tests the condition at the beginning of the loop: The EnterDates1 procedure writes the dates of the current month to a worksheet column, beginning with the active cell.

```
Sub EnterDates1()  
    ' Do While, with test at the beginning  
    Dim TheDate As Date  
    TheDate = DateSerial(Year(Date), Month(Date), 1)  
    Do While Month(TheDate) = Month(Date)  
        ActiveCell = TheDate  
        TheDate = TheDate + 1  
        ActiveCell.Offset(1, 0).Activate  
    Loop  
End Sub
```

This procedure uses a variable, TheDate, which contains the dates that are written to the worksheet. This variable is initialized with the first day of the current month. Inside of the loop, the value of TheDate is entered into the active cell, TheDate is incremented, and the next cell is activated. The loop continues while the month of TheDate is the same as the month of the current date.

The following procedure has the same result as the EnterDates1 procedure, but it uses the second Do While loop syntax, which checks the condition at the end of the loop.

```
Sub EnterDates2()  
    ' Do While, with test at the end  
    Dim TheDate As Date  
    TheDate = DateSerial(Year(Date), Month(Date), 1)  
    Do  
        ActiveCell = TheDate
```



```
TheDate = TheDate + 1
ActiveCell.Offset(1, 0).Activate
Loop While Month(TheDate) = Month(Date)
End Sub
```

The following is another Do While loop example. This procedure opens a text file, reads each line, converts the text to uppercase, and then stores it in the active sheet, beginning with cell A1 and continuing down the column. The procedure uses the VBA EOF function, which returns True when the end of the file has been reached. The final statement closes the text file.

```
Sub DoWhileDemo1()
Dim LineCt As Long
Dim LineOfText As String
Open "c:\data\textfile.txt" For Input As #1
LineCt = 0
Do While Not EOF(1)
Line Input #1, LineOfText
Range("A1").Offset(LineCt, 0) = UCase(LineOfText)
LineCt = LineCt + 1
Loop
Close #1
End Sub
```

Do While loops can also contain one or more Exit Do statements. When an Exit Do statement is encountered, the loop ends immediately and control passes to the statement following the Loop statement.

### Do Until loops

The Do Until loop structure is very similar to the Do While structure. The difference is evident only when the condition is tested. In a Do While loop, the loop executes while the condition is True; in a Do Until loop, the loop executes until the condition is True.

Do Until also has two syntaxes:

```
Do [Until condition]
[instructions]
[Exit Do]
[instructions]
Loop
```

or

```
Do
[instructions]
[Exit Do]
[instructions]
Loop [Until condition]
```

The two examples that follow perform the same action as the Do While date entry examples in the previous section. The difference in these two procedures is where the condition is evaluated (at the beginning or the end of the loop).

```
Sub EnterDates3()  
    ' Do Until, with test at beginning  
    Dim TheDate As Date  
    TheDate = DateSerial(Year(Date), Month(Date), 1)  
    Do Until Month(TheDate) <> Month(Date)  
        ActiveCell = TheDate  
        TheDate = TheDate + 1  
        ActiveCell.Offset(1, 0).Activate  
    Loop  
End Sub
```

```
Sub EnterDates4()  
    ' Do Until, with test at end  
    Dim TheDate As Date  
    TheDate = DateSerial(Year(Date), Month(Date), 1)  
    Do  
        ActiveCell = TheDate  
        TheDate = TheDate + 1  
        ActiveCell.Offset(1, 0).Activate  
    Loop Until Month(TheDate) <> Month(Date)  
End Sub
```

The following example was originally presented for the Do While loop but has been rewritten to use a Do Until loop. The only difference is the line with the Do statement. This example makes the code a bit clearer because it avoids the negative required in the Do While example.

```
Sub DoUntilDemo1()  
    Dim LineCt As Long  
    Dim LineOfText As String  
    Open "c:\data\textfile.txt" For Input As #1  
    LineCt = 0  
    Do Until EOF(1)  
        Line Input #1, LineOfText  
        Range("A1").Offset(LineCt, 0) = UCase(LineOfText)  
        LineCt = LineCt + 1  
    Loop  
    Close #1  
End Sub
```

**Notes: VBA supports yet another type of loop, While Wend. This looping structure is included primarily for compatibility purposes. I mention it here in case you ever encounter such a loop. Here's how the date entry procedure looks when it's coded to use a While Wend loop:**

```
Sub EnterDates5()  
    Dim TheDate As Date  
    TheDate = DateSerial(Year(Date), Month(Date), 1)  
    While Month(TheDate) = Month(Date)  
        ActiveCell = TheDate  
        TheDate = TheDate + 1  
        ActiveCell.Offset(1, 0).Activate  
    Wend  
End Sub
```

### 3.1. About Procedures

## 3. Working with VBA Sub Procedures

A procedure is a series of VBA statements that resides in a VBA module, which you access in the Visual Basic Editor (VBE). A module can hold any number of procedures. A procedure holds a group of VBA statements that accomplishes a desired task. Most VBA code is contained in procedures.

You have a number of ways to call, or execute, procedures. A procedure is executed from beginning to end, but it can also be ended prematurely.

**Notes:** A procedure can be any length, but many people prefer to avoid creating extremely long procedures that perform many different operations. You may find it easier to write several smaller procedures, each with a single purpose. Then, design a main procedure that calls those other procedures. This approach can make your code easier to maintain.

Some procedures are written to receive arguments. An argument is simply information that is used by the procedure and that is passed to the procedure when it is executed. Procedure arguments work much like the arguments that you use in Excel worksheet functions. Instructions within the procedure generally perform logical operations on these arguments, and the results of the procedure are usually based on those arguments.

**Notes:** Although this chapter focuses on Sub procedures, VBA also supports Function procedures, which I discuss in Chapter 10. Chapter 11 has many additional examples of procedures, both Sub and Function, that you can incorporate into your work.

### Declaring a Sub procedure

A procedure declared with the Sub keyword must adhere to the following syntax:

```
[Private | Public][Static] Sub name ([arglist])  
[instructions]  
[Exit Sub]  
[instructions]  
End Sub
```

Here's a description of the elements that make up a Sub procedure:

- **Private:** (Optional) Indicates that the procedure is accessible only to other procedures in the same module.
- **Public:** (Optional) Indicates that the procedure is accessible to all other procedures in all other modules in the workbook. If used in a module that contains an Option Private Module statement, the procedure is not available outside the project.
- **Static:** (Optional) Indicates that the procedure's variables are preserved when the procedure ends.
- **Sub:** (Required) The keyword that indicates the beginning of a procedure.
- **name:** (Required) Any valid procedure name.
- **arglist:** (Optional) Represents a list of variables, enclosed in parentheses, that receive arguments passed to the procedure. Use a

comma to separate arguments. If the procedure uses no arguments, a set of empty parentheses is required.

- **instructions:** (Optional) Represents valid VBA instructions.
- **Exit Sub:** (Optional) A statement that forces an immediate exit from the procedure prior to its formal completion.
- **End Sub:** (Required) Indicates the end of the procedure.

**Notes:** With a few exceptions, all VBA instructions in a module must be contained within procedures. Exceptions include module-level variable declarations, user-defined data type definitions, and a few other instructions that specify module-level options (for example, Option Explicit).

### Naming procedures

Every procedure must have a name. The rules governing procedure names are generally the same as those for variable names. Ideally, a procedure's name should describe what its contained processes do. A good rule is to use a name that includes a verb and a noun (for example, ProcessDate, PrintReport, Sort\_Array, or CheckFilename). Unless you're writing a quick and dirty procedure that you'll use once and delete, avoid meaningless names such as Dolt, Update, and Fix.

Some programmers use sentence-like names that describe the procedure (for example, WriteReportToTextFile and Get\_Print\_Options\_ and\_Print\_Report).

### Scoping a procedure

In the preceding chapter, I note that a variable's scope determines the modules and procedures in which you can use the variable. Similarly, a procedure's scope determines which other procedures can call it.

### Public procedures

By default, procedures are public — that is, they can be called by other procedures in any module in the workbook. It's not necessary to use the Public keyword, but programmers often include it for clarity. The following two procedures are both public:

```
Sub First()  
    ' ... [code goes here] ...  
End Sub  
  
Public Sub Second()  
    ' ... [code goes here] ...  
End Sub
```

### Private procedures

Private procedures can be called by other procedures in the same module but not by procedures in other modules.

**Notes:** When a user displays the Macro dialog box, Excel shows only the public procedures. Therefore, if you have procedures that are designed to be called only by other procedures in the same module, you should make

### 3.2. Executing Sub Procedures

sure that those procedures are declared as **Private**. Doing so prevents the user from running these procedures from the Macro dialog box.

The following example declares a private procedure named MySub:

```
Private Sub MySub()  
    ' ... [code goes here] ...  
End Sub
```

**Notes:** You can force all procedures in a module to be private — even those declared with the **Public** keyword — by including the following statement before your first Sub statement:

```
Option Private Module
```

**If you write this statement in a module, you can omit the Private keyword from your Sub declarations.**

Excel's macro recorder normally creates new Sub procedures called Macro1, Macro2, and so on. Unless you modify the recorded code, these procedures are all public procedures, and they will never use any arguments.

In this section, I describe the various ways to execute, or call, a VBA Sub procedure:

- With the Run⇒Run Sub/UserForm command (in the VBE menu). Or you can press the F5 shortcut key, or click the Run Sub/UserForm button on the Standard toolbar.
- From Excel's Macro dialog box.
- By using the Ctrl key shortcut assigned to the procedure (assuming that you assigned one).
- By clicking a button or a shape on a worksheet. The button or shape must have the procedure assigned to it.
- From another procedure that you write. Sub and Function procedures can execute other procedures.
- From a custom control in the Ribbon. In addition, built-in Ribbon controls can be “repurposed” to execute a macro.
- From a customized shortcut menu.
- When an event occurs. These events include opening the workbook, saving the workbook, closing the workbook, changing a cell's value, activating a sheet, and many other things.
- From the Immediate window in the VBE. Just type the name of the procedure, including any arguments that may apply, and press Enter.

I discuss these methods of executing procedures in the following sections.

**Notes:** In many cases, a procedure won't work properly unless it's executed in the appropriate context. For example, if a procedure is designed to work with the active worksheet, it will fail if a chart sheet is active. A good procedure incorporates code that checks for the appropriate context and exits gracefully if it can't proceed.

**Executing a procedure with the Run Sub/UserForm command**

The VBE Run⇒Run Sub/UserForm menu command is used primarily to test a procedure while you're developing it. You would never require a user to activate the VBE to execute a procedure. Choose Run⇒Run Sub/UserForm in the VBE to execute the current procedure (in other words, the procedure that contains the cursor). Or, press F5, or use the Run Sub/UserForm button on the Standard toolbar.

If the cursor isn't located within a procedure when you issue the Run Sub/UserForm command, VBE displays its Macro dialog box so that you can select a procedure to execute.

### Executing a procedure from the Macro dialog box

Choosing Excel's Developer⇒Code⇒Macros command displays the Macro dialog box, as shown in Figure 9-1. (You can also press Alt+F8 to access this dialog box.) Use the Macros In drop-down box to limit the scope of the macros displayed (for example, show only the macros in the active workbook).

The Macro dialog box does not display

- Function procedures
- Sub procedures declared with the Private keyword
- Sub procedures that require one or more arguments
- Sub procedures contained in add-ins

**Notes:** Even though procedures stored in an add-in are not listed in the Macro dialog box, you still can execute such a procedure if you know the name. Simply type the procedure name in the Macro Name field in the Macro dialog box and then click Run.

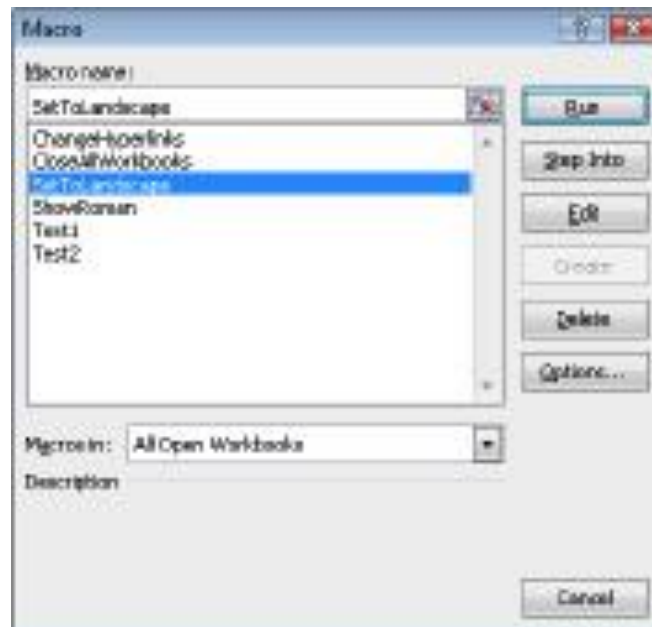


FIGURE 9-1: The Macro dialog box.

### Executing a procedure with a Ctrl+shortcut key combination

You can assign a Ctrl+shortcut key combination to any procedure that doesn't use any arguments. If you assign the Ctrl+U key combo to a procedure named UpdateCustomerList, for example, pressing Ctrl+U executes that procedure.

When you begin recording a macro, the Record Macro dialog box gives you the opportunity to assign a shortcut key. However, you can assign a shortcut key at any time. To assign a Ctrl shortcut key to a procedure (or to change a procedure's shortcut key), follow these steps:

1. Activate Excel and choose Developer⇒Code⇒Macros.
2. Select the appropriate procedure from the list box in the Macro dialog box.
3. Click the Options button to display the Macro Options dialog box (see Figure 9-2).



FIGURE 9-2: The Macro Options dialog box lets you assign a Ctrl key shortcut and an optional description to a procedure.

4. Enter a character into the Ctrl+ text box.
5. Enter a description (optional). If you enter a description for a macro, it's displayed at the bottom of the Macro dialog box when the procedure is selected in the list box.
6. Click OK to close the Macro Options dialog box and then click Cancel to close the Macro dialog box.

**Notes:** If you assign one of Excel's predefined shortcut key combinations to a procedure, your key assignment takes precedence over the predefined key assignment. For example, Ctrl+S is the Excel predefined shortcut key for saving the active workbook. But if you assign Ctrl+S to a procedure, pressing Ctrl+S no longer saves the active workbook.

### Executing a procedure from the Ribbon

Excel's Ribbon user interface was introduced in Excel 2007. In that version, customizing the Ribbon required writing XML code to add a new button (or other control) to the Ribbon. Note that you modify the Ribbon in this way outside of Excel, and you can't do it using VBA.

### Executing a procedure from a customized shortcut menu

You can also execute a macro by clicking a menu item in a customized shortcut menu. A shortcut menu appears when you right-click an object or range in Excel.

### Executing a procedure from another procedure

One of the most common ways to execute a procedure is from another VBA procedure. You have three ways to do this:

- Enter the procedure's name, followed by its arguments (if any) separated by commas.



- Use the Call keyword followed by the procedure's name and then its arguments (if any) enclosed in parentheses and separated by commas.
- Use the Run method of the Application object. The Run method is useful when you need to run a procedure whose name is assigned to a variable. You can then pass the variable as an argument to the Run method.

The following example demonstrates the first method. In this case, the MySub procedure processes some statements (not shown), executes the UpdateSheet procedure, and then executes the rest of the statements.

```
Sub MySub()  
    ' ... [code goes here] ...  
    UpdateSheet  
    ' ... [code goes here] ...  
End Sub  
  
Sub UpdateSheet()  
    ' ... [code goes here] ...  
End Sub
```

The following example demonstrates the second method. The Call keyword executes the Update procedure, which requires one argument; the calling procedure passes the argument to the called procedure. I discuss procedure arguments later in this chapter (see "Passing Arguments to Procedures").

```
Sub MySub()  
    MonthNum = InputBox("Enter the month number: ")  
    Call UpdateSheet(MonthNum)  
    ' ... [code goes here] ...  
End Sub  
  
Sub UpdateSheet(MonthSeq)  
    ' ... [code goes here] ...  
End Sub
```

The next example uses the Run method to execute the UpdateSheet procedure and then to pass MonthNum as the argument.

```
Sub MySub()  
    MonthNum = InputBox("Enter the month number: ")  
    Application.Run "UpdateSheet", MonthNum  
    ' ... [code goes here] ...  
End Sub  
  
Sub UpdateSheet(MonthSeq)  
    ' ... [code goes here] ...  
End Sub
```

Perhaps the best reason to use the Run method is when the procedure name is assigned to a variable. In fact, it's the only way to execute a procedure in such a way. The following example demonstrates this. The Main procedure uses the VBA WeekDay function to determine the day of the week (an integer between 1 and 7, beginning with Sunday). The SubToCall variable is assigned a string that

represents a procedure name. The Run method then calls the appropriate procedure (either WeekEnd or Daily).

```
Sub Main()  
Dim SubToCall As String  
Select Case WeekDay(Now)  
Case 1, 7: SubToCall = "WeekEnd"  
Case Else: SubToCall = "Daily"  
End Select  
Application.Run SubToCall  
End Sub  
  
Sub WeekEnd()  
MsgBox "Today is a weekend"  
' Code to execute on the weekend  
' goes here  
End Sub  
  
Sub Daily()  
MsgBox "Today is not a weekend"  
' Code to execute on the weekdays  
' goes here  
End Sub
```

### Calling a procedure in a different module

If VBA can't locate a called procedure in the current module, it looks for public procedures in other modules in the same project.

If you need to call a private procedure from another procedure, both procedures must reside in the same module.

You can't have two procedures with the same name in the same module, but you can have identically named procedures in different modules within the project. You can force VBA to execute an ambiguously named procedure — that is, another procedure in a different module that has the same name. To do so, precede the procedure name with the module name and a dot. For example, say that you define procedures named MySub in Module1 and Module2. If you want a procedure in Module2 to call the MySub in Module1, you can use either of the following statements:

```
Module1.MySub  
Call Module1.MySub
```

If you do not differentiate between procedures that have the same name, you get an Ambiguous name detected error message.

### Calling a procedure in a different workbook

In some cases, you may need your procedure to execute another procedure defined in a different workbook. To do so, you have two options: Either establish a reference to the other workbook or use the Run method and specify the workbook name explicitly.

To add a reference to another workbook, choose the VBE's Tools⇒References command. Excel displays the References dialog box (see Figure 9-3), which lists all available references, including all open workbooks. Simply check the box that corresponds to the workbook that you want to add as a reference and then click OK. After you establish a reference, you can call procedures in the workbook as if they were in the same workbook as the calling procedure.

A referenced workbook doesn't have to be open when you create the reference; it's treated like a separate object library. Use the Browse button in the References dialog box to establish a reference to a workbook that isn't open.

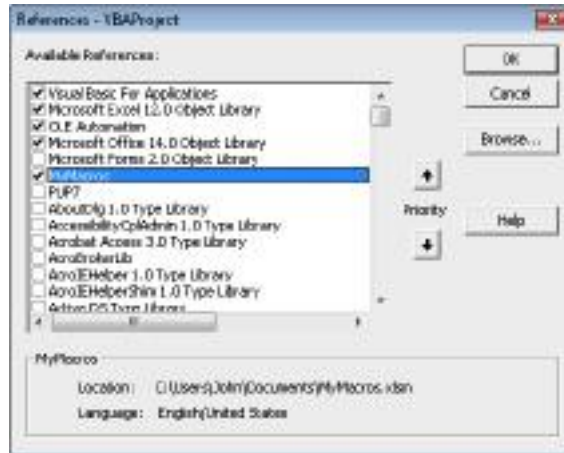


FIGURE 9-3: The References dialog box lets you establish a reference to another workbook.

When you open a workbook that contains a reference to another workbook, the referenced workbook is opened automatically.

The list of references displayed in the References dialog box also includes object libraries and ActiveX controls that are registered on your system. Your Excel workbooks always include references to the following object libraries:

- Visual Basic for Applications
- Microsoft Excel 14.0 Object Library
- OLE Automation
- Microsoft Office 14.0 Object Library
- Microsoft Forms 2.0 Object Library (this reference is included only if your project includes a UserForm)

**Notes:** Any additional references to other workbooks that you add are also listed in your project outline in the Project Explorer window in the VBE. These references are listed under a node called References.

If you've established a reference to a workbook that contains the procedure MySub, for example, you can use either of the following statements to call MySub:

*YourSub*

*Call YourSub*

To precisely identify a procedure in a different workbook, specify the project name, module name, and procedure name by using the following syntax:

*MyProject.MyModule.MySub*

Alternatively, you can use the Call keyword:

*Call MyProject.MyModule.MySub*

Another way to call a procedure in a different workbook is to use the Run method of the Application object. This technique doesn't require that you establish a reference, but the workbook that contains the procedure must be open. The following statement executes the Consolidate procedure located in a workbook named budget macros.xlsm:

```
Application.Run "budget macros.xlsm"!Consolidate"
```

### **Why call other procedures?**

If you're new to programming, you may wonder why anyone would ever want to call a procedure from another procedure. You may ask, "Why not just put the code from the called procedure into the calling procedure and keep things simple?"

One reason is to clarify your code. The simpler your code, the easier it is to maintain and modify. Smaller routines are easier to decipher and then debug. Examine the accompanying procedure, which does nothing but call other procedures. This procedure is very easy to follow.

```
Sub Main()  
    Call GetUserOptions  
    Call ProcessData  
    Call CleanUp  
    Call CloseItDown  
End Sub
```

Calling other procedures also eliminates redundancy. Suppose that you need to perform an operation at ten different places in your routine. Rather than enter the code ten times, you can write a procedure to perform the operation and then simply call the procedure ten times. Also, if you need to make a change, you make it only one time rather than ten times.

Also, you may have a series of general-purpose procedures that you use frequently. If you store these in a separate module, you can import the module to your current project and then call these procedures as needed — which is much easier than copying and pasting the code into your new procedures.

Creating several small procedures rather than a single large one is often considered good programming practice. A modular approach not only makes your job easier but also makes life easier for the people who wind up working with your code.

### **Executing a procedure by clicking an object**

Excel provides a variety of objects that you can place on a worksheet or chart sheet, and you can attach a macro to any of these objects. These objects fall into several classes:

- ActiveX controls
- Forms controls
- Inserted objects (Shapes, SmartArt, WordArt, charts, and pictures)

To assign a procedure to a Button object from the Form controls, follow these steps:

1. Select Developer⇒Controls⇒Insert and click the button icon in the Form Controls group.

2. Click the worksheet to create the button.

Or, you can drag your mouse on the worksheet to change the default size of the button.

Excel jumps right in and displays the Assign Macro dialog box (see Figure 9-4). It proposes a macro that's based on the button's name.

3. Select or enter the macro that you want to assign to the button and then click OK.

You can always change the macro assignment by right-clicking the button and choosing Assign Macro.

To assign a macro to a Shape, SmartArt, WordArt, chart, or picture, right-click the object and choose Assign Macro from the shortcut menu.

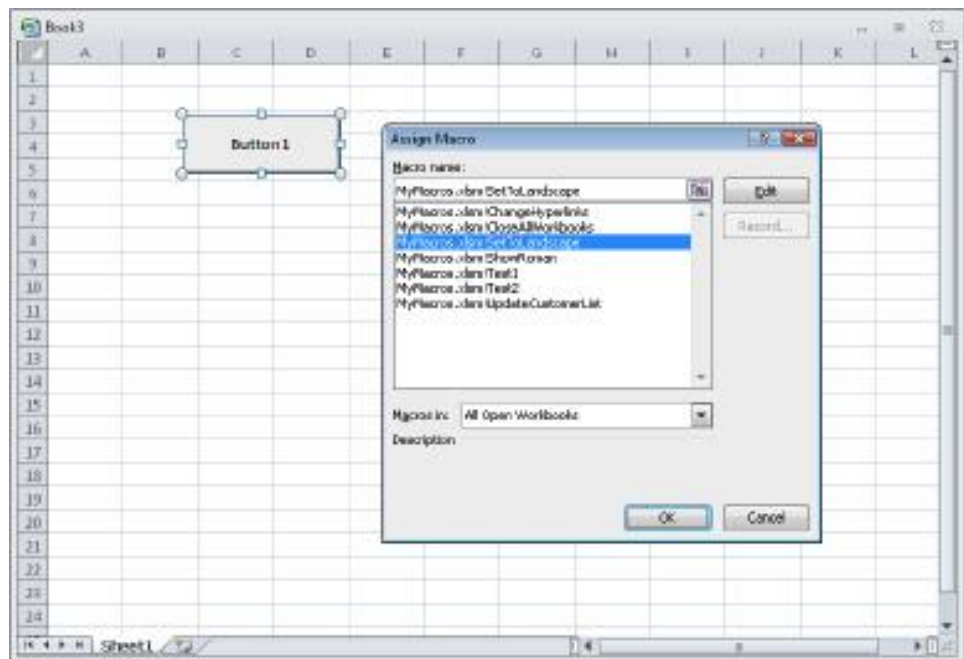


FIGURE 9-4: Assigning a macro to a button.

### Executing a procedure when an event occurs

You might want a procedure to execute when a particular event occurs. Examples of events include opening a workbook, entering data into a worksheet, saving a workbook, clicking a CommandButton ActiveX control, and many others. A procedure that is executed when an event occurs is an event handler procedure. Event handler procedures are characterized by the following:

- They have special names that are made up of an object, an underscore, and the event name. For example, the procedure that is executed when a workbook is opened is `Workbook_Open`.
- They're stored in the Code module for the particular object.

### Executing a procedure from the Immediate window

You also can execute a procedure by entering its name in the Immediate window of the VBE. If the Immediate window isn't visible, press `Ctrl+G`. The Immediate window executes VBA statements while you enter them. To execute a procedure, simply enter the name of the procedure in the Immediate window and press `Enter`.

This method can be quite useful when you're developing a procedure because you can insert commands to display results in the Immediate window. The following procedure demonstrates this technique:

```
Sub ChangeCase()  
    Dim MyString As String  
    MyString = "This is a test"  
    MyString = UCase(MyString)  
    Debug.Print MyString  
End Sub
```

Figure 9-5 shows what happens when you enter ChangeCase in the Immediate window: The Debug.Print statement displays the result immediately.

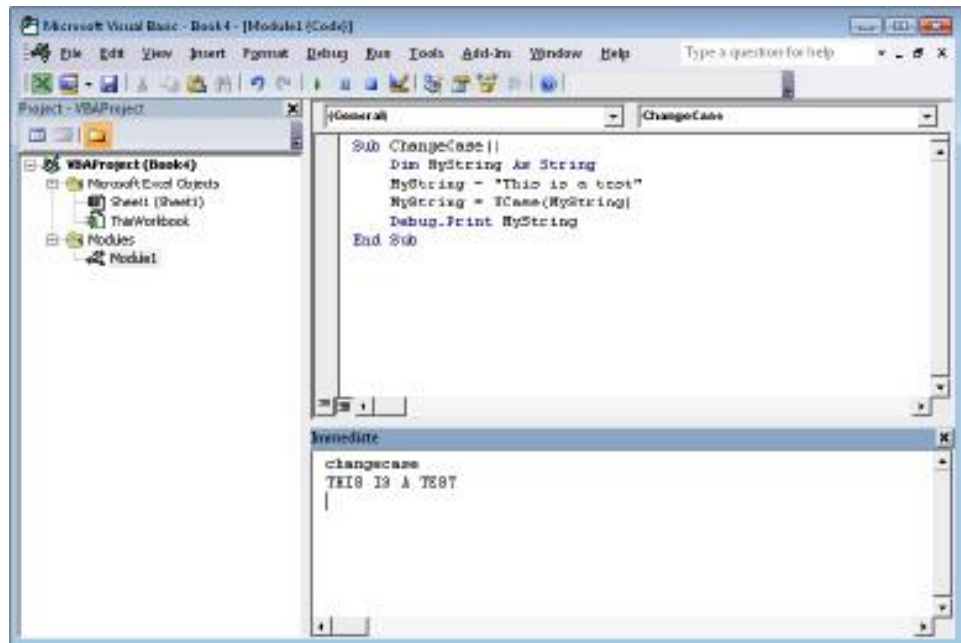


FIGURE 9-5: Executing a procedure by entering its name in the Immediate window.

### 3.3. Passing Arguments to Procedures

A procedure's arguments provide it with data that it uses in its instructions. The data that's passed by an argument can be any of the following:

- A variable
- A constant
- An array
- An object

The use of arguments by procedures is very similar to their use of worksheet functions in the following respects:

- A procedure may not require any arguments.
- A procedure may require a fixed number of arguments.
- A procedure may accept an indefinite number of arguments.
- A procedure may require some arguments, leaving others optional.
- A procedure may have all optional arguments.

For example, a few of Excel's worksheet functions, such as RAND and NOW, use no arguments. Others, such as COUNTIF, require two arguments. Others

still, such as SUM, can use up to 255 arguments. Still other worksheet functions have optional arguments. The PMT function, for example, can have five arguments (three are required; two are optional).

Most of the procedures that you've seen so far in this book have been declared without arguments. They were declared with just the Sub keyword, the procedure's name, and a set of empty parentheses. Empty parentheses indicate that the procedure does not accept arguments.

The following example shows two procedures. The Main procedure calls the ProcessFile procedure three times (the Call statement is in a For-Next loop). Before calling ProcessFile, however, a three-element array is created. Inside the loop, each element of the array becomes the argument for the procedure call. The ProcessFile procedure takes one argument (named TheFile). Notice that the argument goes inside parentheses in the Sub statement. When ProcessFile finishes, program control continues with the statement after the Call statement.

```
Sub Main()  
    Dim File(1 To 3) As String  
    Dim i as Integer  
    File(1) = "dept1.xlsx"  
    File(2) = "dept2.xlsx"  
    File(3) = "dept3.xlsx"  
    For i = 1 To 3  
        Call ProcessFile(File(i))  
    Next i  
End Sub  
  
Sub ProcessFile(TheFile)  
    Workbooks.Open FileName:=TheFile  
    ' ...[more code here]...  
End Sub
```

You can also, of course, pass literals (that is, not variables) to a procedure. For example:

```
Sub Main()  
    Call ProcessFile("budget.xlsx")  
End Sub
```

You can pass an argument to a procedure in two ways:

- By reference: Passing an argument by reference simply passes the memory address of the variable. Changes to the argument within the procedure are made to the original variable. This is the default method of passing an argument.
- By value: Passing an argument by value passes a copy of the original variable. Consequently, changes to the argument within the procedure are not reflected in the original variable.

The following example demonstrates this concept. The argument for the Process procedure is passed by reference (the default method). After the Main procedure assigns a value of 10 to MyValue, it calls the Process procedure and passes MyValue as the argument. The Process procedure multiplies the value of its argument (named YourValue) by 10. When Process ends and program control passes back to Main, the MsgBox function displays MyValue: 100.

```
Sub Main()
```

```
Dim MyValue As Integer
MyValue = 10
Call Process(MyValue)
MsgBox MyValue
End Sub

Sub Process(YourValue)
YourValue = YourValue * 10
End Sub
```

If you don't want the called procedure to modify any variables passed as arguments, you can modify the called procedure's argument list so that arguments are passed to it by value rather than by reference. To do so, precede the argument with the `ByVal` keyword. This technique causes the called routine to work with a copy of the passed variable's data — not the data itself. In the following procedure, for example, the changes made to `YourValue` in the `Process` procedure do not affect the `MyValue` variable in `Main`. As a result, the `MsgBox` function displays 10 and not 100.

```
Sub Process(ByVal YourValue)
YourValue = YourValue * 10
End Sub
```

In most cases, you'll be content to use the default reference method of passing arguments. However, if your procedure needs to use data passed to it in an argument — and you must keep the original data intact — you'll want to pass the data by value.

A procedure's arguments can mix and match by value and by reference. Arguments preceded with `ByVal` are passed by value; all others are passed by reference.

**If you pass a variable defined as a user-defined data type to a procedure, it must be passed by reference. Attempting to pass it by value generates an error.**

Because I didn't declare a data type for any of the arguments in the preceding examples, all the arguments have been of the `Variant` data type. But a procedure that uses arguments can define the data types directly in the argument list. The following is a `Sub` statement for a procedure with two arguments of different data types. The first is declared as an integer, and the second is declared as a string.

```
Sub Process(Iterations As Integer, TheFile As String)
```

### Using public variables versus passing arguments to a procedure

In Chapter 8, I point out how a variable declared as `Public` (at the top of the module) is available to all procedures in the module. In some cases, you may want to access a `Public` variable rather than pass the variable as an argument when calling another procedure.

For example, the procedure that follows passes the value of `MonthVal` to the `ProcessMonth` procedure:

```
Sub MySub()
Dim MonthVal as Integer
' ... [code goes here]
MonthVal = 4
```



### 3.4. Error-Handling Techniques

```
Call ProcessMonth(MonthVal)
```

```
' ... [code goes here]
```

```
End Sub
```

An alternative approach, which doesn't use an argument, is

```
Public MonthVal as Integer
```

```
Sub MySub()
```

```
' ... [code goes here]
```

```
MonthVal = 4
```

```
Call ProcessMonth2
```

```
' ... [code goes here]
```

```
End Sub
```

In the revised code, because MonthVal is a public variable, the ProcessMonth2 procedure can access it, thus eliminating the need for an argument for the ProcessMonth2 procedure.

When you pass arguments to a procedure, the data that is passed as the argument must match the argument's data type. For example, if you call Process in the preceding example and pass a string variable for the first argument, you get an error: ByRef argument type mismatch.

When a VBA procedure is running, errors can (and probably will) occur. These include either syntax errors (which you must correct before you can execute a procedure) or runtime errors (which occur while the procedure is running). This section deals with runtime errors.

**Notes: for error-handling procedures to work, the Break on All Errors setting must be turned off. In the VBE, choose Tools⇒Options and click the General tab in the Options dialog box. If Break on All Errors is selected, VBA ignores your error-handling code. You'll usually want to use the Break on Unhandled Errors option.**

Normally, a runtime error causes VBA to stop, and the user sees a dialog box that displays the error number and a description of the error. A good application doesn't make the user deal with these messages. Rather, it incorporates error-handling code to trap errors and take appropriate actions. At the very least, your error-handling code can display a more meaningful error message than the one VBA pops up.

You can use the On Error statement to specify what happens when an error occurs. Basically, you have two choices:

- Ignore the error and let VBA continue. Your code can later examine the Err object to determine what the error was and then take action, if necessary.
- Jump to a special error-handling section of your code to take action. This section is placed at the end of the procedure and is also marked by a label.

To cause your VBA code to continue when an error occurs, insert the following statement in your code:

```
On Error Resume Next
```

Some errors are inconsequential, and you can ignore them without causing a problem. But you might want to determine what the error was. When an error occurs, you can use the Err object to determine the error number. You can use

the VBA Error function to display the text that corresponds to the Err.Number value. For example, the following statement displays the same information as the normal Visual Basic error dialog box (the error number and the error description):

```
MsgBox "Error " & Err & ": " & Error(Err.Number)
```

Figure 9-6 shows a VBA error message, and Figure 9-7 shows the same error displayed in a message box. You can, of course, make the error message a bit more meaningful to your end users by using more descriptive text.

**Notes: Referencing Err is equivalent to accessing the Number property of the Err object. Therefore, the following two statements have the same effect:**

```
MsgBox Err
```

```
MsgBox Err.Number
```

You also use the On Error statement to specify a location in your procedure to jump to when an error occurs. You use a label to mark the location. For example:

```
On Error GoTo ErrorHandler
```

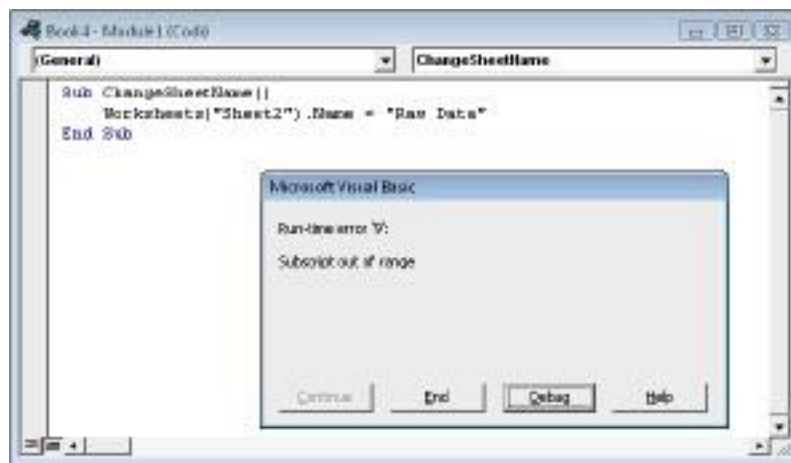


FIGURE 9-6: VBA error messages aren't always user friendly.

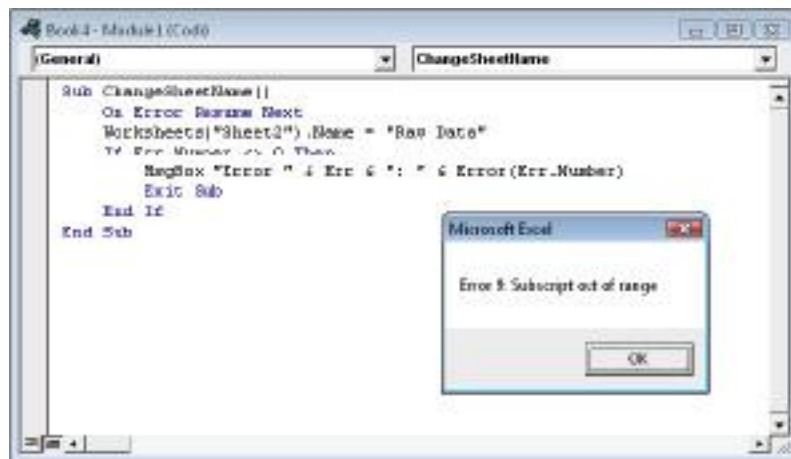


FIGURE 9-7: You can create a message box to display the error code and description.

### Error-handling examples

The first example demonstrates an error that you can safely ignore. The SpecialCells method selects cells that meet a certain criterion.

**Notes:** The `SpecialCells` method is equivalent to choosing the **Home**⇒**Editing**⇒**Find & Select**⇒**Go To Special** command. The **Go To Special** dialog box provides you with a number of choices. For example, you can select cells that contain a numeric constant (nonformula).

In the example that follows, which doesn't use any error handling, the `SpecialCells` method selects all the cells in the current range selection that contain a formula that returns a number. If no cells in the selection qualify, VBA displays the error message shown in Figure 9-8.



FIGURE 9-8: The `SpecialCells` method generates this error if no cells are found.

```
Sub SelectFormulas()
    Selection.SpecialCells(xlFormulas, xlNumbers).Select
    '...[more code goes here]
End Sub
```

Following is a variation that uses the `On Error Resume Next` statement to prevent the error message from appearing:

```
Sub SelectFormulas2()
    On Error Resume Next
    Selection.SpecialCells(xlFormulas, xlNumbers).Select
    On Error GoTo 0
    '...[more code goes here]
End Sub
```

The `On Error GoTo 0` statement restores normal error handling for the remaining statements in the procedure.

The following procedure uses an additional statement to determine whether an error did occur. If so, the user is informed by a message.

```
Sub SelectFormulas3()
    On Error Resume Next
    Selection.SpecialCells(xlFormulas, xlNumbers).Select
    If Err.Number = 1004 Then MsgBox "No formula cells were found."
    On Error GoTo 0
    '...[more code goes here]
End Sub
```

If the `Number` property of `Err` is equal to anything other than 0, then an error occurred. The `If` statement checks to see if `Err.Number` is equal to 1004 and displays a message box if it is. In this example, the code is checking for a specific error number. To check for any error, use a statement like this:

```
If Err.Number <> 0 Then MsgBox "An error occurred."
```

The next example demonstrates error handling by jumping to a label.

```
Sub ErrorDemo()  
On Error GoTo Handler  
Selection.Value = 123  
Exit Sub  
Handler:  
MsgBox "Cannot assign a value to the selection."  
End Sub
```

The procedure attempts to assign a value to the current selection. If an error occurs (for example, a range isn't selected or the sheet is protected), the assignment statement results in an error. The On Error statement specifies a jump to the Handler label if an error occurs. Notice the use of the Exit Sub statement before the label. This statement prevents the error-handling code from being executed if no error occurs. If this statement is omitted, the error message is displayed even if an error does not occur.

Sometimes, you can take advantage of an error to get information. The example that follows simply checks whether a particular workbook is open. It doesn't use any error handling.

```
Sub CheckForFile1()  
Dim FileName As String  
Dim FileExists As Boolean  
Dim book As Workbook  
FileName = "BUDGET.XLSX"  
FileExists = False  
' Cycle through all open workbooks  
For Each book In Workbooks  
If UCase(book.Name) = FileName Then FileExists = True  
Next book  
' Display appropriate message  
If FileExists Then  
MsgBox FileName & " is open."  
Else  
MsgBox FileName & " is not open."  
End If  
End Sub
```

Here, a For Each-Next loop cycles through all objects in the Workbooks collection. If the workbook is open, the FileExists variable is set to True. Finally, a message is displayed that tells the user whether the workbook is open.

You can rewrite the preceding routine to use error handling to determine whether the file is open. In the example that follows, the On Error Resume Next statement causes VBA to ignore any errors. The next instruction attempts to reference the workbook by assigning the workbook to an object variable (by using the Set keyword). If the workbook isn't open, an error occurs. The If-Then-Else structure checks the value property of Err and displays the appropriate message. This procedure uses no looping, so it's slightly more efficient.

```
Sub CheckForFile()
```

```
Dim FileName As String
Dim x As Workbook
FileName = "BUDGET.XLSX"
On Error Resume Next
Set x = Workbooks(FileName)
If Err = 0 Then
MsgBox FileName & " is open."
Else
MsgBox FileName & " is not open."
End If
On Error GoTo 0
End Sub
```

### 3.5. A Realistic Example That Uses Sub Procedures

In this chapter, I describe the basics of creating Sub procedures. Most of the previous examples, I will admit, have been rather wimpy. The remainder of this chapter is a real-life exercise that demonstrates many of the concepts covered in this and the preceding two chapters.

This section describes the development of a useful utility that qualifies as an application as defined in Chapter 5. More important, I demonstrate the process of analyzing a problem and then solving it with VBA. I wrote this section with VBA newcomers in mind. As a result, I don't simply present the code, but I also show how to find out what you need to know to develop the code.

#### The goal

The goal of this exercise is to develop a utility that rearranges a workbook by alphabetizing its sheets (something that Excel can't do on its own). If you tend to create workbooks that consist of many sheets, you know that locating a particular sheet can be difficult. If the sheets are ordered alphabetically, however, it's easier to find a desired sheet.

#### Project requirements

Where to begin? One way to get started is to list the requirements for your application. When you develop your application, you can check your list to ensure that you're covering all the bases.

Here's the list of requirements that I compiled for this example application:

1. It should sort the sheets (that is, worksheets and chart sheets) in the active workbook in ascending order of their names.
2. It should be easy to execute.
3. It should always be available. In other words, the user shouldn't have to open a workbook to use this utility.
4. It should work properly for any workbook that's open.
5. It should not display any VBA error messages.

**What you know**

Often, the most difficult part of a project is figuring out where to start. In this case, I started by listing things that I know about Excel that may be relevant to the project requirements:

- Excel doesn't have a command that sorts sheets, so I'm not re-inventing the wheel.
- I can't create this type of macro by recording my actions. However, the macro might be useful to provide some key information.
- I can move a sheet easily by dragging its sheet tab.
- Mental note: Turn on the macro recorder and drag a sheet to a new location to find out what kind of code this action generates.
- Excel also has a Move or Copy dialog box, which is displayed when I right-click a sheet tab and choose Move or Copy. Would recording a macro of this command generate different code than moving a sheet manually?
- I'll need to know how many sheets are in the active workbook. I can get this information with VBA.
- I'll need to know the names of all the sheets. Again, I can get this information with VBA.
- Excel has a command that sorts data in worksheet cells.
- Mental note: Maybe I can transfer the sheet names to a range and use this feature. Or, maybe VBA has a sorting method that I can take advantage of.
- Thanks to the Macro Options dialog box, it's easy to assign a shortcut key to a macro.
- If a macro is stored in the Personal Macro Workbook, it will always be available.
- I need a way to test the application while I develop it. For certain, I don't want to be testing it using the same workbook in which I'm developing the code.
- If I develop the code properly, VBA won't display any errors.

**The approach**

Although I still didn't know exactly how to proceed, I could devise a preliminary, skeleton plan that describes the general tasks required:

1. Identify the active workbook.
2. Get a list of all the sheet names in the workbook.
3. Count the sheets.
4. Sort the sheet names (somehow).
5. Rearrange the sheets so they correspond to the sorted sheet names.

**What you need to know**

I saw a few holes in the plan. I knew that I had to determine the following:

- How to identify the active workbook
- How to count the sheets in the active workbook
- How to get a list of the sheet names

- How to sort the list
- How to rearrange the sheets according to the sorted list

### Some preliminary recording

Here's an example of using the macro recorder to learn about VBA. I started with a workbook that contained three worksheets. Then I turned on the macro recorder and specified my Personal Macro Workbook as the destination for the macro. With the macro recorder running, I dragged the third worksheet to the first sheet position. Here's the code that was generated by the macro recorder:

```
Sub Macro1()  
    Sheets("Sheet3").Select  
    Sheets("Sheet3").Move Before:=Sheets(1)  
End Sub
```

I searched the VBA Help for Move and discovered that it's a method that moves a sheet to a new location in the workbook. It also takes an argument that specifies the location for the sheet. This information is very relevant to the task at hand. Curious, I then turned on the macro recorder to see whether using the Move or Copy dialog box would generate different code. It didn't.

Next, I needed to find out how many sheets were in the active workbook. I searched Help for the word Count and found out that it's a property of a collection. I activated the Immediate window in the VBE and typed the following statement:

```
? ActiveWorkbook.Count
```

Error! After a little more thought, I realized that I needed to get a count of the sheets within a workbook. So I tried this:

```
? ActiveWorkbook.Sheets.Count
```

Success. Figure 9-9 shows the result. More useful information.



FIGURE 9-9: Use the VBE Immediate window to test a statement.

What about the sheet names? Time for another test. I entered the following statement in the Immediate window:

```
? ActiveWorkbook.Sheets(1).Name
```

This told me that the name of the first sheet is Sheet3, which is correct (because I'd moved it). More good information to keep in mind.

Then I remembered something about the For Each-Next construct: It's useful for cycling through each member of a collection. After consulting the Help system, I created a short procedure to test it:

```
Sub Test()  
    For Each Sht In ActiveWorkbook.Sheets  
        MsgBox Sht.Name  
    Next Sht  
End Sub
```

Another success. This macro displayed three message boxes, each showing a different sheet name.

Finally, it was time to think about sorting options. From the Help system, I learned that the Sort method applies to a Range object. So one option was to transfer the sheet names to a range and then sort the range, but that seemed like overkill for this application. I thought that a better option was to dump the sheet names into an array of strings and then sort the array by using VBA code.

### Initial setup

Now I knew enough to get started writing some serious code. Before doing so, however, I needed to do some initial setup work. To re-create my steps, follow these instructions:

1. Create an empty workbook with five worksheets, named Sheet1, Sheet2, Sheet3, Sheet4, and Sheet5.
2. Move the sheets around randomly so that they aren't in any particular order.
3. Save the workbook as Test.xlsx.
4. Activate the VBE and select the Personal.xlsm project in the Project Window.

If Personal.xlsm doesn't appear in the Project window in the VBE, it means that you've never used the Personal Macro Workbook. To have Excel create this workbook for you, simply record a macro (any macro) and specify the Personal Macro Workbook as the destination for the macro.

5. Insert a new VBA module in Personal.xlsm (choose Insert⇒Module).
6. Create an empty Sub procedure called SortSheets (see Figure 9-10).

Actually, you can store this macro in any module in the Personal Macro Workbook. However, keeping each group of related macros in a separate module is a good idea. That way, you can easily export the module and import it into a different project later on.

7. Activate Excel and choose Developer⇒Code⇒Macros to display the Macro dialog box.
8. In the Macro dialog box, select the SortSheets procedure and click the Options button to assign a shortcut key to this macro.

The Ctrl+Shift+S key combination is a good choice.



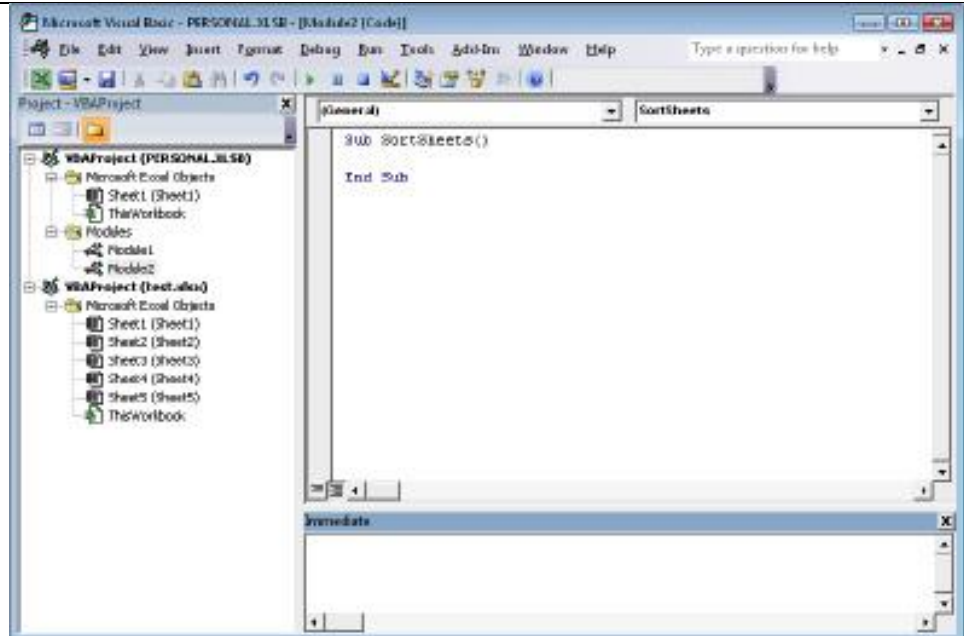


FIGURE 9-10: An empty procedure in a module located in the Personal Macro Workbook.

### Code writing

Now it's time to write some code. I knew that I needed to put the sheet names into an array of strings. Because I didn't know yet how many sheets were in the active workbook, I used a Dim statement with empty parentheses to declare the array. I knew that I could use ReDim afterward to redimension the array for the actual number of elements.

I entered the following code, which inserted the sheet names into the SheetNames array. I also added a MsgBox function within the loop just to assure me that the sheets' names were indeed being entered into the array.

```
Sub SortSheets()
    ' Sorts the sheets of the active workbook
    Dim SheetNames() as String
    Dim i as Long
    Dim SheetCount as Long
    SheetCount = ActiveWorkbook.Sheets.Count
    ReDim SheetNames(1 To SheetCount)
    For i = 1 To SheetCount
        SheetNames(i) = ActiveWorkbook.Sheets(i).Name
        MsgBox SheetNames(i)
    Next i
End Sub
```

To test the preceding code, I activated the Test.xlsx workbook and pressed Ctrl+Shift+S. Five message boxes appeared, each displaying the name of a sheet in the active workbook. So far, so good.

By the way, I'm a major proponent of testing your work as you go. I tend to work in small steps and set things up so that I'm convinced that each small step is working properly before I continue. When you're convinced that your code is

working correctly, remove the MsgBox statement. (These message boxes become annoying after a while.)

**Notes: Rather than use the MsgBox function to test your work, you can use the Print method of the Debug object to display information in the Immediate window. For this example, use the following statement in place of the MsgBox statement:**

*Debug.Print SheetNames(i)*

**This technique is much less intrusive than using MsgBox statements. Just make sure that you remember to remove the statement when you're finished.**

At this point, the SortSheets procedure simply creates an array of sheet names corresponding to the sheets in the active workbook. Two steps remain: Sort the elements in the SheetNames array and then rearrange the sheets to correspond to the sorted array.

### Writing the Sort procedure

It was time to sort the SheetNames array. One option was to insert the sorting code in the SortSheets procedure, but I thought a better approach was to write a general-purpose sorting procedure that I could reuse with other projects. (Sorting arrays is a common operation.)

You might be a bit daunted by the thought of writing a sorting procedure. The good news is that it's relatively easy to find commonly used routines that you can use or adapt. The Internet, of course, is a great source for such information.

You can sort an array in many ways. I chose the bubble sort method; although it's not a particularly fast technique, it's easy to code. Blazing speed isn't really a requirement in this particular application.

The bubble sort method uses a nested For-Next loop to evaluate each array element. If the array element is greater than the next element, the two elements swap positions. The code includes a nested loop, so this evaluation is repeated for every pair of items (that is,  $n - 1$  times).

Here's the sorting procedure I developed (after consulting a few programming Web sites to get some ideas):

```
Sub BubbleSort(List() As String)
    ' Sorts the List array in ascending order
    Dim First As Long, Last As Long
    Dim i As Long, j As Long
    Dim Temp As String
    First = LBound(List)
    Last = UBound(List)
    For i = First To Last - 1
        For j = i + 1 To Last
            If List(i) > List(j) Then
                Temp = List(j)
                List(j) = List(i)
                List(i) = Temp
            End If
        Next j
    Next i
End Sub
```

```
End If
```

```
Next j
```

```
Next i
```

```
End Sub
```

This procedure accepts one argument: a one-dimensional array named List. An array passed to a procedure can be of any length. I used the LBound and UBound functions to assign the lower bound and upper bound of the array to the variables First and Last, respectively.

Here's a little temporary procedure that I used to test the BubbleSort procedure:

```
Sub SortTester()
```

```
Dim x(1 To 5) As String
```

```
Dim i As Long
```

```
x(1) = "dog"
```

```
x(2) = "cat"
```

```
x(3) = "elephant"
```

```
x(4) = "aardvark"
```

```
x(5) = "bird"
```

```
Call BubbleSort(x)
```

```
For i = 1 To 5
```

```
Debug.Print i, x(i)
```

```
Next i
```

```
End Sub
```

The SortTester routine creates an array of five strings, passes the array to BubbleSort, and then displays the sorted array in the Immediate window. I eventually deleted this code because it served its purpose.

After I was satisfied that this procedure worked reliably, I modified SortSheets by adding a call to the BubbleSort procedure, passing the SheetNames array as an argument. At this point, my module looked like this:

```
Sub SortSheets()
```

```
Dim SheetNames() As String
```

```
Dim SheetCount as Long
```

```
Dim i as Long
```

```
SheetCount = ActiveWorkbook.Sheets.Count
```

```
ReDim SheetNames(1 To SheetCount)
```

```
For i = 1 To SheetCount
```

```
SheetNames(i) = ActiveWorkbook.Sheets(i).Name
```

```
Next i
```

```
Call BubbleSort(SheetNames)
```

```
End Sub
```

```
Sub BubbleSort(List() As String)
```

```
‘ Sorts the List array in ascending order
```

```
Dim First As Long, Last As Long
```

```

    Dim i As Long, j As Long
    Dim Temp As String
    First = LBound(List)
    Last = UBound(List)
    For i = First To Last - 1
        For j = i + 1 To Last
            If List(i) > List(j) Then
                Temp = List(j)
                List(j) = List(i)
                List(i) = Temp
            End If
        Next j
    Next i
End Sub

```

When the SheetSort procedure ends, it contains an array that consists of the sorted sheet names in the active workbook. To verify this, you can display the array contents in the VBE Immediate window by adding the following code at the end of the SortSheets procedure (if the Immediate window is not visible, press Ctrl+G):

```

    For i = 1 To SheetCount
        Debug.Print SheetNames(i)
    Next i

```

So far, so good. Next step: Write some code to rearrange the sheets to correspond to the sorted items in the SheetNames array.

The code that I recorded earlier proved useful. Remember the instruction that was recorded when I moved a sheet to the first position in the workbook?

```

Sheets("Sheet3").Move Before:=Sheets(1)

```

After a little thought, I was able to write a For-Next loop that would go through each sheet and move it to its corresponding sheet location, specified in the SheetNames array:

```

    For i = 1 To SheetCount
        Sheets(SheetNames(i)).Move Before:=Sheets(i)
    Next i

```

For example, the first time through the loop, the loop counter i is 1. The first element in the SheetNames array is (in this example) Sheet1. Therefore, the expression for the Move method within the loop evaluates to

```

Sheets("Sheet1").Move Before:= Sheets(1)

```

The second time through the loop, the expression evaluates to

```

Sheets("Sheet2").Move Before:= Sheets(2)

```

I then added the new code to the SortSheets procedure:

```

Sub SortSheets()
    Dim SheetNames() As String
    Dim SheetCount as Long
    Dim i as Long

```

```
SheetCount = ActiveWorkbook.Sheets.Count
ReDim SheetNames(1 To SheetCount)
For i = 1 To SheetCount
    SheetNames(i) = ActiveWorkbook.Sheets(i).Name
Next i
Call BubbleSort(SheetNames)
For i = 1 To SheetCount
    ActiveWorkbook.Sheets(SheetNames(i)).Move _
        Before:=ActiveWorkbook.Sheets(i)
Next i
End Sub
```

I did some testing, and it seemed to work just fine for the Test.xlsx workbook.

Time to clean things up. I made sure that all the variables used in the procedures were declared, and then I added a few comments and blank lines to make the code easier to read. The SortSheets procedure looked like the following:

```
Sub SortSheets()
    ' This routine sorts the sheets of the
    ' active workbook in ascending order.
    ' Use Ctrl+Shift+S to execute
    Dim SheetNames() As String
    Dim SheetCount As Long
    Dim i As Long
    ' Determine the number of sheets & ReDim array
    SheetCount = ActiveWorkbook.Sheets.Count
    ReDim SheetNames(1 To SheetCount)
    ' Fill array with sheet names
    For i = 1 To SheetCount
        SheetNames(i) = ActiveWorkbook.Sheets(i).Name
    Next i
    ' Sort the array in ascending order
    Call BubbleSort(SheetNames)
    ' Move the sheets
    For i = 1 To SheetCount
        ActiveWorkbook.Sheets(SheetNames(i)).Move _
            Before:= ActiveWorkbook.Sheets(i)
    Next i
End Sub
```

Everything seemed to be working. To test the code further, I added a few more sheets to Test.xlsx and changed some of the sheet names. It worked like a charm.

**More testing**

I was tempted to call it a day. However, just because the procedure worked with the Test.xlsx workbook didn't mean that it would work with all workbooks. To test it further, I loaded a few other workbooks and retried the routine. I soon discovered that the application wasn't perfect. In fact, it was far from perfect. I identified the following problems:

- Workbooks with many sheets took a long time to sort because the screen was continually updated during the move operations.
- The sorting didn't always work. For example, in one of my tests, a sheet named SUMMARY (all uppercase) appeared before a sheet named Sheet1. This problem was caused by the BubbleSort procedure — an uppercase U is “greater than” a lowercase h.
- If Excel had no visible workbook windows, pressing the Ctrl+Shift+S shortcut key combo caused the macro to fail.
- If the workbook's structure was protected, the Move method failed.
- After sorting, the last sheet in the workbook became the active sheet. Changing the user's active sheet isn't a good practice; it's better to keep the user's original sheet active.
- If I interrupted the macro by pressing Ctrl+Break, VBA displayed an error message.
- The macro can't be reversed (that is, the Undo command is always disabled when a macro is executed). If the user accidentally presses Ctrl+Shift+S, the workbook sheets are sorted, and the only way to get them back to their original order is by doing it manually.

**Fixing the problems**

Fixing the screen-updating problem was a breeze. I inserted the following instruction to turn off screen updating while the sheets were being moved:

*Application.ScreenUpdating = False*

This statement causes Excel's windows to freeze while the macro is running. A beneficial side effect is that it also speeds up the macro considerably. After the macro completes its operation, screen updating is turned back on automatically.

It was also easy to fix the problem with the BubbleSort procedure: I used VBA's UCase function to convert the sheet names to uppercase for the comparison. This caused all the comparisons to be made by using uppercase versions of the sheet names. The corrected line read as follows:

*If UCase(List(i)) > UCase(List(j)) Then*

**Notes: Another way to solve the “case” problem is to add the following statement to the top of your module:**

*Option Compare Text*

**This statement causes VBA to perform string comparisons based on a case-insensitive text sort order. In other words, A is considered the same as a.**

To prevent the error message that appears when no workbooks are visible, I added some error checking. I used On Error Resume Next to ignore the error and then checked the value of Err. If Err is not equal to 0, it means that an error occurred. Therefore, the procedure ends. The error-checking code is

*On Error Resume Next*

*SheetCount = ActiveWorkbook.Sheets.Count*

*If Err <> 0 Then Exit Sub ' No active workbook*

It occurred to me that I could avoid using On Error Resume Next. The following statement is a more direct approach to determining whether a workbook isn't visible and doesn't require any error handling. This statement can go at the top of the SortSheets procedure:

*If ActiveWorkbook Is Nothing Then Exit Sub*

There's usually a good reason that a workbook's structure is protected. I decided that the best approach was to not attempt to unprotect the workbook. Rather, the code should display a message box warning and let the user unprotect the workbook and re-execute the macro. Testing for a protected workbook structure was easy — the ProtectStructure property of a Workbook object returns True if a workbook is protected. I added the following block of code:

*' Check for protected workbook structure*

*If ActiveWorkbook.ProtectStructure Then*

*MsgBox ActiveWorkbook.Name & " is protected.", \_*

*vbCritical, "Cannot Sort Sheets."*

*Exit Sub*

*End If*

If the workbook's structure is protected, the user sees a message box like the one shown in Figure 9-11.

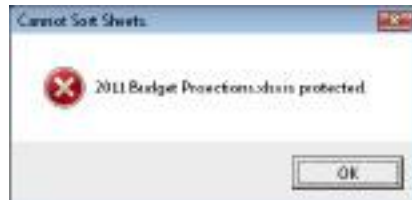


FIGURE 9-11: This message box tells the user that the sheets cannot be sorted.

To reactivate the original active sheet after the sorting was performed, I wrote code that assigned the original sheet to an object variable (OldActiveSheet) and then activated that sheet when the routine was finished. Here's the statement that assigns the variable:

*Set OldActive = ActiveSheet*

This statement activates the original active worksheet:

*OldActive.Activate*

Pressing Ctrl+Break normally halts a macro, and VBA usually displays an error message. But because one of my goals was to avoid VBA error messages, I inserted a command to prevent this situation. From the VBA Help system, I discovered that the Application object has an EnableCancelKey property that can disable Ctrl+Break. So I added the following statement at the top of the routine:

*Application.EnableCancelKey = xlDisabled*

**Notes:** Be very careful when you disable the Cancel key. If your code gets caught in an infinite loop, you can't break out of it. For best results, insert this statement only after you're sure that everything is working properly.

To prevent the problem of accidentally sorting the sheets, I added the following statement to the procedure, before the Ctrl+Break key is disabled:

```
If MsgBox("Sort the sheets in the active workbook?", _
vbQuestion + vbYesNo) <> vbYes Then Exit Sub
```

When the user executes the SortSheets procedure, he sees the message box in Figure 9-12.



FIGURE 9-12: This message box appears before the sheets are sorted.

After I made all these corrections, the SortSheets procedure looked like this:

```
Option Explicit
Sub SortSheets()
' This routine sorts the sheets of the
' active workbook in ascending order.
' Use Ctrl+Shift+S to execute
Dim SheetNames() As String
Dim i As Long
Dim SheetCount As Long
Dim OldActiveSheet As Object
If ActiveWorkbook Is Nothing Then Exit Sub ' No active workbook
SheetCount = ActiveWorkbook.Sheets.Count
' Check for protected workbook structure
If ActiveWorkbook.ProtectStructure Then
MsgBox ActiveWorkbook.Name & " is protected.", _
vbCritical, "Cannot Sort Sheets."
Exit Sub
End If

' Make user verify
If MsgBox("Sort the sheets in the active workbook?", _
vbQuestion + vbYesNo) <> vbYes Then Exit Sub
' Disable Ctrl+Break
Application.EnableCancelKey = xlDisabled
' Get the number of sheets
SheetCount = ActiveWorkbook.Sheets.Count
' Redimension the array
ReDim SheetNames(1 To SheetCount)
' Store a reference to the active sheet
Set OldActiveSheet = ActiveSheet
' Fill array with sheet names
```



```
For i = 1 To SheetCount
    SheetNames(i) = ActiveWorkbook.Sheets(i).Name
Next i

' Sort the array in ascending order
Call BubbleSort(SheetNames)

' Turn off screen updating
Application.ScreenUpdating = False

' Move the sheets
For i = 1 To SheetCount
    ActiveWorkbook.Sheets(SheetNames(i)).Move _
    Before:=ActiveWorkbook.Sheets(i)
Next i

' Reactivate the original active sheet
OldActiveSheet.Activate

End Sub
```

### Utility availability

Because the SortSheets macro is stored in the Personal Macro Workbook, it's available whenever Excel is running. At this point, you can execute the macro by selecting the macro's name from the Macro dialog box (Alt+F8 displays this dialog box) or by pressing Ctrl+Shift+S. Another option is to add a command to the Ribbon.

To add a command, follow these steps:

1. Right-click any area of the Ribbon and choose Customize the Ribbon.
2. In the Customize Ribbon tab of the Excel Options dialog box, choose Macros from the drop-down list labeled Choose Commands From.
3. Click the item labeled PERSONAL.XLSB!SortSheets.
4. Use the controls in the box on the right to specify the ribbon tab and create a new group.

(You can't add a command to an existing group.)

I created a group named Worksheets in the View tab, and renamed the new item to Short Sheets (see Figure 9-13).

### Evaluating the project

So there you have it. The utility meets all the original project requirements: It sorts all sheets in the active workbook, it can be executed easily, it's always available, it seems to work for any workbook, and I have yet to see it display a VBA error message.

**Notes: The procedure still has one slight problem: The sorting is strict and may not always be "logical." For example, after sorting, Sheet10 is placed before Sheet2. Most would want Sheet2 to be listed before Sheet10. Solving that problem is beyond the scope of this introductory exercise.**

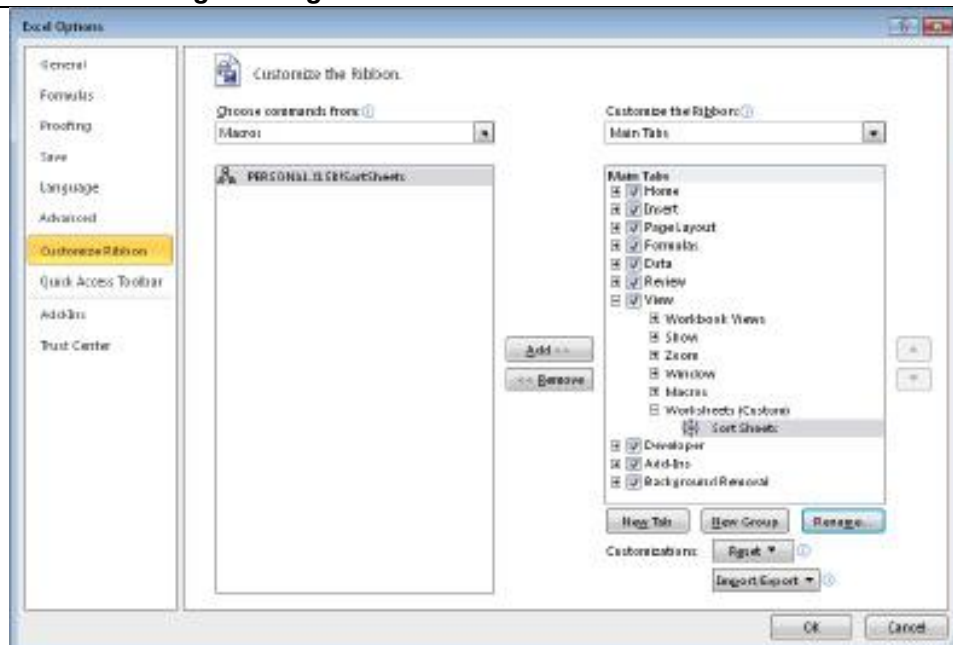


FIGURE 9-13: Adding a new command to the ribbon.

## 4. Creating Function Procedures

### 4.1. Sub Procedures versus Function Procedures

A function is a VBA procedure that performs calculations and returns a value. You can use these functions in your Visual Basic for Applications (VBA) code or in formulas.

VBA enables you to create Sub procedures and Function procedures. You can think of a Sub procedure as a command that either the user or another procedure can execute. Function procedures, on the other hand, usually return a single value (or an array), just like Excel worksheet functions and VBA built-in functions. As with built-in functions, your Function procedures can use arguments.

Function procedures are quite versatile, and you can use them in two situations:

- As part of an expression in a VBA procedure
- In formulas that you create in a worksheet

In fact, you can use a Function procedure anywhere that you can use an Excel worksheet function or a VBA built-in function. As far as I know, the only exception is that you can't use a VBA function in a data validation formula.

I cover Sub procedures in the preceding chapter, and in this chapter, I discuss Function procedures.

### 4.2. Why Create Custom Functions?

You're undoubtedly familiar with Excel worksheet functions; even novices know how to use the most common worksheet functions, such as SUM, AVERAGE, and IF. Excel 2010 includes more than 400 predefined worksheet functions that you can use in formulas. If that's not enough, however, you can create custom functions by using VBA.

With all the functions available in Excel and VBA, you might wonder why you'd ever need to create new functions. The answer: to simplify your work. With a bit of planning, custom functions are very useful in worksheet formulas and VBA procedures.

Often, for example, you can create a custom function that can significantly shorten your formulas. And shorter formulas are more readable and easier to work with. I should also point out, however, that custom functions used in your formulas are usually much slower than built-in functions. And, of course, the user must enable macros in order to use these functions.

When you create applications, you may notice that some procedures repeat certain calculations. In such cases, consider creating a custom function that performs the calculation. Then you can simply call the function from your procedure. A custom function can eliminate the need for duplicated code, thus reducing errors.

Also, coworkers often can benefit from your specialized functions. And some may be willing to pay you to create custom functions that save them time and work.

Although many cringe at the thought of creating custom worksheet functions, the process isn't difficult. In fact, I enjoy creating custom functions. I especially like how my custom functions appear in the Insert Function dialog box along with Excel built-in functions, as if I'm re-engineering the software in some way.

In this chapter, I tell you what you need to know to start creating custom functions, and I provide lots of examples.

### 4.3. An Introductory Function Example

Without further ado, this section presents an example of a VBA Function procedure.

The following is a custom function defined in a VBA module. This function, named RemoveVowels, uses a single argument. The function returns the argument, but with all the vowels removed.

```
Function RemoveVowels(Txt) As String
    ' Removes all vowels from the Txt argument
    Dim i As Long
    RemoveVowels = ""
    For i = 1 To Len(Txt)
        If Not UCase(Mid(Txt, i, 1)) Like "[AEIOU]" Then
            RemoveVowels = RemoveVowels & Mid(Txt, i, 1)
        End If
    Next i
End Function
```

This function certainly isn't the most useful one I've written, but it demonstrates some key concepts related to functions. I explain how this function works later, in the "Analyzing the custom function" section.

**Notes: When you create custom functions that will be used in a worksheet formula, make sure that the code resides in a normal VBA module. If you place your custom functions in a code module for a UserForm, a Sheet, or ThisWorkbook, they won't work in your formulas. Your formulas will return a #NAME? error.**

#### Using the function in a worksheet

When you enter a formula that uses the RemoveVowels function, Excel executes the code to get the value. Here's an example of how you'd use the function in a formula:

```
=RemoveVowels(A1)
```

See Figure 10-1 for examples of this function in action. The formulas are in column B, and they use the text in column A as their arguments. As you can see, the function returns the single argument, but with the vowels removed.

Actually, the function works pretty much like any built-in worksheet function. You can insert it in a formula by choosing Formulas⇒Function Library⇒Insert Function or by clicking the Insert Function Wizard icon to the left of the formula bar. Either of these actions displays the Insert Function dialog box. In the Insert Function dialog box, your custom functions are located, by default, in the User Defined category.

You can also nest custom functions and combine them with other elements in your formulas. For example, the following formula nests the RemoveVowels function inside Excel's UPPER function. The result is the original string (sans vowels), converted to uppercase.

```
=UPPER(RemoveVowels(A1))
```

	A	B
1	Every good boy does fine.	vry gd by ds fn.
2	antidisestablishmentarianism	ntdtsblshmntnm
3	Microsoft Excel	Mcrft xcl
4	abcdefghijklmnopqrstuvwxyz	bcdfghjklmnpqrstvwyz
5	A failure to communicate.	flr tmmnt.
6	This sentence has no vowels.	Ths mntc hs n vcls.
7		
8		
9		

FIGURE 10-1: Using a custom function in a worksheet formula.

### Using the function in a VBA procedure

In addition to using custom functions in worksheet formulas, you can use them in other VBA procedures. The following VBA procedure, which is defined in the same module as the custom RemoveVowels function, first displays an input box to solicit text from the user. Then the procedure uses the VBA built-in MsgBox function to display the user input after the RemoveVowels function processes it (see Figure 10-2). The original input appears as the caption in the message box.

```
Sub ZapTheVowels()
    Dim UserInput as String
    UserInput = InputBox("Enter some text:")
    MsgBox RemoveVowels(UserInput), vbInformation, UserInput
End Sub
```

In the example shown in Figure 10-2, the string entered in response to the InputBox function was Excel 2010 Power Programming With VBA. The MsgBox function displays the text without vowels.



FIGURE 10-2: Using a custom function in a VBA procedure.

### Analyzing the custom function

Function procedures can be as complex as you need them to be. Most of the time, they're more complex and much more useful than this sample procedure. Nonetheless, an analysis of this example may help you understand what is happening.

Here's the code, again:

```
Function RemoveVowels(Txt) As String
    ' Removes all vowels from the Txt argument
    Dim i As Long
    RemoveVowels = ""
    For i = 1 To Len(Txt)
        If Not UCase(Mid(Txt, i, 1)) Like "[AEIOU]" Then
            RemoveVowels = RemoveVowels & Mid(Txt, i, 1)
        End If
    Next i
End Function
```

```
Next i
```

```
End Function
```

Notice that the procedure starts with the keyword Function, rather than Sub, followed by the name of the function (RemoveVowels). This custom function uses only one argument (Txt), enclosed in parentheses. As String defines the data type of the function's return value. Excel uses the Variant data type if no data type is specified.

The second line is an optional comment that describes what the function does. This line is followed by a Dim statement, which declares the variable (i) used in the procedure as type Long.

**Notes: Notice that I use the function name as a variable here. When a function ends, it always returns the current value of the variable that corresponds to the function's name.**

The next five instructions make up a For-Next loop. The procedure loops through each character in the input and builds the string. The first instruction within the loop uses VBA's Mid function to return a single character from the input string and converts this character to uppercase. That character is then compared to a list of characters by using VBA's Like operator. In other words, the If clause is true if the character isn't A, E, I, O, or U. In such a case, the character is appended to the RemoveVowels variable.

When the loop is finished, RemoveVowels consists of the input string with all the vowels removed. This string is the value that the function returns.

The procedure ends with an End Function statement.

Keep in mind that you can do the coding for this function in a number of different ways. Here's a function that accomplishes the same result but is coded differently:

```
Function RemoveVowels(txt) As String
```

```
    ' Removes all vowels from the Txt argument
```

```
    Dim i As Long
```

```
    Dim TempString As String
```

```
    TempString = ""
```

```
    For i = 1 To Len(txt)
```

```
        Select Case ucase(Mid(txt, i, 1))
```

```
            Case "A", "E", "I", "O", "U"
```

```
                'Do nothing
```

```
            Case Else
```

```
                TempString = TempString & Mid(txt, i, 1)
```

```
        End Select
```

```
    Next i
```

```
    RemoveVowels = TempString
```

```
End Function
```

In this version, I used a string variable (TempString) to store the vowel-less string as it's being constructed. Then, before the procedure ends, I assigned the contents of TempString to the function's name. This version also uses a Select Case construct rather than an If-Then construct.

#### 4.4. Function Procedures

##### What custom worksheet functions can't do

When you develop custom functions, it's important to understand a key distinction between functions that you call from other VBA procedures and functions that you use in worksheet formulas. Function procedures used in worksheet formulas must be passive. For example, code within a Function procedure can't manipulate ranges or change things on the worksheet. An example can help make this limitation clear.

You may be tempted to write a custom worksheet function that changes a cell's formatting. For example, it may be useful to have a formula that uses a custom function to change the color of text in a cell based on the cell's value. Try as you might, however, such a function is impossible to write. No matter what you do, the function won't change the worksheet. Remember, a function simply returns a value. It can't perform actions with objects.

That said, I should point out one notable exception. You can change the text in a cell comment by using a custom VBA function. I'm not sure if this behavior is intentional, or if it's a bug in Excel. In any case, modifying a comment via a function seems to work reliably. Here's the function:

```
Function ModifyComment(Cell As Range, Cmt As String)
```

```
Cell.Comment.Text Cmt
```

```
End Function
```

Here's an example of using this function in a formula. The formula replaces the comment in cell A1 with new text. The function won't work if cell A1 doesn't have a comment.

```
=ModifyComment(A1,"Hey, I changed your comment")
```

A custom Function procedure has much in common with a Sub procedure. (For more information on Sub procedures, see Chapter 9.)

The syntax for declaring a function is as follows:

```
[Public | Private][Static] Function name ([arglist])[As type]
```

```
[instructions]
```

```
[name = expression]
```

```
[Exit Function]
```

```
[instructions]
```

```
[name = expression]
```

```
End Function
```

The Function procedure contains the following elements:

- **Public:** (Optional) Indicates that the Function procedure is accessible to all other procedures in all other modules in all active Excel VBA projects.
- **Private:** (Optional) Indicates that the Function procedure is accessible only to other procedures in the same module.
- **Static:** (Optional) Indicates that the values of variables declared in the Function procedure are preserved between calls.
- **Function:** (Required) Indicates the beginning of a procedure that returns a value or other data.
- **name:** (Required) Represents any valid Function procedure name, which must follow the same rules as a variable name.
- **arglist:** (Optional) Represents a list of one or more variables that represent arguments passed to the Function procedure. The arguments

are enclosed in parentheses. Use a comma to separate pairs of arguments.

- **type:** (Optional) Is the data type returned by the Function procedure.
- **instructions:** (Optional) Are any number of valid VBA instructions.
- **Exit Function:** (Optional) Is a statement that forces an immediate exit from the Function procedure prior to its completion.
- **End Function:** (Required) Is a keyword that indicates the end of the Function procedure.

A key point to remember about a custom function written in VBA is that a value is always assigned to the function's name a minimum of one time, generally when it has completed execution.

To create a custom function, start by inserting a VBA module. You can use an existing module, as long as it's a normal VBA module. Enter the keyword **Function**, followed by the function name and a list of its arguments (if any) in parentheses. You can also declare the data type of the return value by using the **As** keyword (this is optional, but recommended). Insert the VBA code that performs the work, making sure that the appropriate value is assigned to the term corresponding to the function name at least once within the body of the Function procedure. End the function with an **End Function** statement.

Function names must adhere to the same rules as variable names. If you plan to use your custom function in a worksheet formula, be careful if the function name is also a cell address. For example, if you use something like J21 as a function name, you can't use the function in a worksheet formula.

The best advice is to avoid using function names that are also cell references, including named ranges. And, avoid using function names that correspond to Excel's built-in function names. In the case of a function name conflict, Excel always uses its built-in function.

### A function's scope

In Chapter 9, I discuss the concept of a procedure's scope (public or private). The same discussion applies to functions: A function's scope determines whether it can be called by procedures in other modules or in worksheets.

Here are a few things to keep in mind about a function's scope:

- If you don't declare a function's scope, its default is **Public**.
- Functions declared **As Private** don't appear in Excel's Insert Function dialog box. Therefore, when you create a function that should be used only in a VBA procedure, you should declare it **Private** so that users don't try to use it in a formula.
- If your VBA code needs to call a function that's defined in another workbook, set up a reference to the other workbook by choosing the Visual Basic Editor (VBE) **Tools⇒References** command.

### Executing function procedures

Although you can execute a Sub procedure in many ways, you can execute a Function procedure in only four ways:

- Call it from another procedure.
- Use it in a worksheet formula.
- Use it in a formula that's used to specify conditional formatting.
- Call it from the VBE Immediate window.



**From a procedure**

You can call custom functions from a VBA procedure the same way that you call built-in functions. For example, after you define a function called `SumArray`, you can enter a statement like the following:

*Total = SumArray(MyArray)*

This statement executes the `SumArray` function with `MyArray` as its argument, returns the function's result, and assigns it to the `Total` variable.

You also can use the `Run` method of the `Application` object. Here's an example:

*Total = Application.Run ("SumArray", "MyArray")*

The first argument for the `Run` method is the function name. Subsequent arguments represent the argument(s) for the function. The arguments for the `Run` method can be literal strings (as shown above), numbers, or variables.

**In a worksheet formula**

Using custom functions in a worksheet formula is like using built-in functions except that you must ensure that Excel can locate the Function procedure. If the Function procedure is in the same workbook, you don't have to do anything special. If it's in a different workbook, you may have to tell Excel where to find it.

You can do so in three ways:

- Precede the function name with a file reference. For example, if you want to use a function called `CountNames` that's defined in an open workbook named `Myfuncs.xlsm`, you can use the following reference:

*=Myfuncs.xlsm!CountNames(A1:A1000)*

If you insert the function with the `Insert Function` dialog box, the workbook reference is inserted automatically.

- Set up a reference to the workbook. You do so by choosing the `VBE Tools⇒References` command. If the function is defined in a referenced workbook, you don't need to use the worksheet name. Even when the dependent workbook is assigned as a reference, the `Paste Function` dialog box continues to insert the workbook reference (although it's not necessary).
- Create an add-in. When you create an add-in from a workbook that has Function procedures, you don't need to use the file reference when you use one of the functions in a formula. The add-in must be installed, however. I discuss add-ins in Chapter 21.

You'll notice that unlike Sub procedures, your Function procedures don't appear in the `Macro` dialog box when you issue the `Developer⇒Code⇒Macros` command. In addition, you can't choose a function when you issue the `VBE Run⇒Sub/UserForm` command (or press `F5`) if the cursor is located in a Function procedure. (You get the `Macro` dialog box that lets you choose a macro to run.) As a result, you need to do a bit of extra up-front work to test your functions while you're developing them. One approach is to set up a simple procedure that calls the function. If the function is designed to be used in worksheet formulas, you'll want to enter a simple formula to test it.

**In a conditional formatting formula**

When you specify conditional formatting, one of the options is to create a formula. The formula must be a logical formula (that is, it must return either

TRUE or FALSE). If the formula returns TRUE, the condition is met, and formatting is applied to the cell.

You can use custom VBA functions in your conditional formatting formulas. For example, here's a simple VBA function that returns TRUE if its argument is a cell that contains a formula:

```
Function CELLHASFORMULA(cell) As Boolean
    CELLHASFORMULA = cell.HasFormula
End Function
```

After defining this function in a VBA module, you can set up a conditional formatting rule so that cells that contain a formula contain different formatting:

1. Select the range that will contain the conditional formatting.  
For example, select A1:G20.
2. Choose Home⇒Styles⇒Conditional Formatting⇒New Rule.
3. In the New Formatting Rule dialog box, select the option labeled Use a Formula to Determine Which Cells to Format.
4. Enter this formula in the formula box — but make sure that the cell reference argument corresponds to the upper-left cell in the range that you selected in Step 1:  
`=CELLHASFORMULA(A1)`
5. Click the Format button to specify the formatting for cells that meet this condition.
6. Click OK to apply the conditional formatting rule to the selected range.

Cells in the range that contain a formula will display the formatting you specified. Figure 10-3 shows the New Formatting Rule dialog box, specifying a custom function in a formula.



FIGURE 10-3: Using a custom VBA function for conditional formatting.

### From the VBE Immediate Window

The final way to call a Function procedure is from the VBE Immediate window. This method is generally used only for testing purposes. Figure 10-4 shows an example.

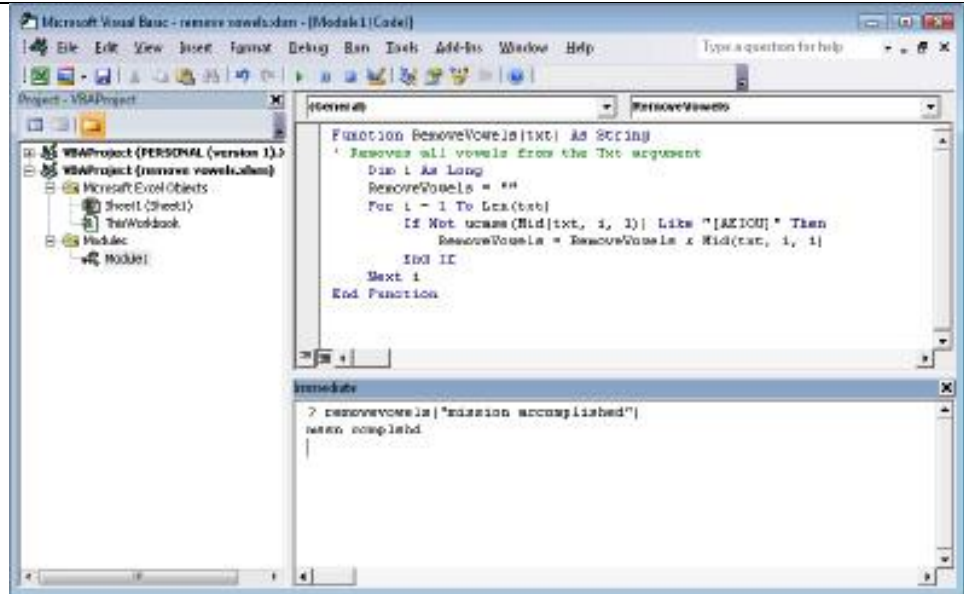


FIGURE 10-4: Calling a Function procedure from the Immediate window.

#### 4.5. Function Arguments

Keep in mind the following points about Function procedure arguments:

- Arguments can be variables (including arrays), constants, literals, or expressions.
- Some functions don't have arguments.
- Some functions have a fixed number of required arguments (from 1 to 60).
- Some functions have a combination of required and optional arguments.

#### Reinventing the wheel

Just for fun, I wrote my own version of Excel's UPPER function (which converts a string to all uppercase) and named it UpCase:

```
Function UpCase(InString As String) As String
```

```
    ' Converts its argument to all uppercase.
```

```
    Dim StringLength As Integer
```

```
    Dim i As Integer
```

```
    Dim ASCIIVal As Integer
```

```
    Dim CharVal As Integer
```

```
    StringLength = Len(InString)
```

```
    UpCase = InString
```

```
    For i = 1 To StringLength
```

```
        ASCIIVal = Asc(Mid(InString, i, 1))
```

```
        CharVal = 0
```

```
        If ASCIIVal >= 97 And ASCIIVal <= 122 Then
```

```
            CharVal = -32
```

```
            Mid(UpCase, i, 1) = Chr(ASCIIVal + CharVal)
```

```
        End If
```

*Next i*

*End Function*

Note: A workbook that contains this function is on the companion CD-ROM in a file named upper case.xlsm.

Notice that I resisted the urge to take the easy route — using the VBA UCase function.

I was curious to see how the custom function differed from the built-in function, so I created a worksheet that called the function 20,000 times, using random names. The worksheet took about 20 seconds to calculate. I then substituted Excel's UPPER function and ran the test again. The recalculation time was virtually instantaneous. I don't claim that my UpCase function is the optimal algorithm for this task, but it's safe to say that a custom function will never match the speed of Excel's built-in functions.

For another example of reinventing the wheel, see “Emulating Excel's SUM Function,” later in this chapter.

## 5. VBA Programming Examples and Techniques

### 5.1. Working with Ranges

The examples in this section demonstrate how to manipulate worksheet ranges with VBA.

Specifically, I provide examples of copying a range, moving a range, selecting a range, identifying types of information in a range, prompting for a cell value, determining the first empty cell in a column, pausing a macro to allow the user to select a range, counting cells in a range, looping through the cells in a range, and several other commonly used range-related operations.

#### Copying a range

Excel's macro recorder is useful not so much for generating usable code, but for discovering the names of relevant objects, methods, and properties. The code that's generated by the macro recorder isn't always the most efficient, but it can usually provide you with several clues.

For example, recording a simple copy-and-paste operation generates five lines of VBA code:

```
Sub Macro1()  
    Range("A1").Select  
    Selection.Copy  
    Range("B1").Select  
    ActiveSheet.Paste  
    Application.CutCopyMode = False  
End Sub
```

#### Using the examples in this chapter

Not all the examples in this chapter are intended to be stand-alone programs. They are, however, set up as usable procedures that you can adapt for your own applications.

I urge you to follow along on your computer as you read this chapter. Better yet, modify the examples and see what happens. I guarantee that this hands-on experience will help you more than reading a reference book.

Notice that the generated code selects cell A1, copies it, and then selects cell B1 and performs the paste operation. But in VBA, you don't need to select an object to work with it. You would never learn this important point by mimicking the preceding recorded macro code, where two statements incorporate the Select method. You can replace this procedure with the following much simpler routine, which doesn't select any cells. It also takes advantage of the fact that the Copy method can use an argument that represents the destination for the copied range.

```
Sub CopyRange()  
    Range("A1").Copy Range("B1")  
End Sub
```

Both of these macros assume that a worksheet is active and that the operation takes place on the active worksheet. To copy a range to a different worksheet or

workbook, simply qualify the range reference for the destination. The following example copies a range from Sheet1 in File1.xlsx to Sheet2 in File2.xlsx. Because the references are fully qualified, this example works regardless of which workbook is active.

```
Sub CopyRange2()  
Workbooks("File1.xlsx").Sheets("Sheet1").Range("A1").Copy _  
Workbooks("File2.xlsx").Sheets("Sheet2").Range("A1")  
End Sub
```

Another way to approach this task is to use object variables to represent the ranges, as shown in the code that follows:

```
Sub CopyRange3()  
Dim Rng1 As Range, Rng2 As Range  
Set Rng1 = Workbooks("File1.xlsx").Sheets("Sheet1").Range("A1")  
Set Rng2 = Workbooks("File2.xlsx").Sheets("Sheet2").Range("A1")  
Rng1.Copy Rng2  
End Sub
```

As you might expect, copying isn't limited to one single cell at a time. The following procedure, for example, copies a large range. Notice that the destination consists of only a single cell (which represents the upper-left cell for the destination). Using a single cell for the destination works just like it does when you copy and paste a range manually in Excel.

```
Sub CopyRange4()  
Range("A1:C800").Copy Range("D1")  
End Sub
```

### Moving a range

The VBA instructions for moving a range are very similar to those for copying a range, as the following example demonstrates. The difference is that you use the Cut method instead of the Copy method. Note that you need to specify only the upper-left cell for the destination range.

The following example moves 18 cells (in A1:C6) to a new location, beginning at cell H1:

```
Sub MoveRange1()  
Range("A1:C6").Cut Range("H1")  
End Sub
```

### Copying a variably sized range

In many cases, you need to copy a range of cells, but you don't know the exact row and column dimensions of the range. For example, you might have a workbook that tracks weekly sales, and the number of rows changes weekly when you add new data.

Figure 11-1 shows a common type of worksheet. This range consists of several rows, and the number of rows changes each week. Because you don't know the exact range address at any given time, writing a macro to copy the range requires additional coding.

	A	B	C	D
1	Week	Total Sales	New Customers	
2	1	21,093	45	
3	2	25,375	49	
4	3	20,180	38	
5	4	25,564	32	
6	5	29,325	22	
7	6	23,009	25	
8	7	24,281	53	
9				

FIGURE 11-1: The number of rows in the data range changes every week.

The following macro demonstrates how to copy this range from Sheet1 to Sheet2 (beginning at cell A1). It uses the CurrentRegion property, which returns a Range object that corresponds to the block of cells around a particular cell (in this case, A1).

```
Sub CopyCurrentRegion2()
    Range("A1").CurrentRegion.Copy Sheets("Sheet2").Range("A1")
End Sub
```

If the range to be copied is a table (specified using Insert⇒Tables⇒Table), you can use code like this (which assumes the table is named Table1):

```
Sub CopyTable()
    Range("Table1[#All]").Copy Sheets("Sheet2").Range("A1")End Sub
```

### Selecting or otherwise identifying various types of ranges

Much of the work that you'll do in VBA will involve working with ranges — either selecting a range or identifying a range so that you can do something with the cells.

In addition to the CurrentRegion property (which I discussed earlier), you should also be aware of the End method of the Range object. The End method takes one argument, which determines the direction in which the selection is extended. The following statement selects a range from the active cell to the last non-empty cell:

```
Range(ActiveCell, ActiveCell.End(xlDown)).Select
```

Here's a similar example that uses a specific cell as the starting point:

```
Range(Range("A2"), Range("A2").End(xlDown)).Select
```

As you might expect, three other constants simulate key combinations in the other directions: xlUp, xlToLeft, and xlToRight.

The following macro is in the example workbook. The SelectCurrentRegion macro simulates pressing Ctrl+Shift+\*.

```
Sub SelectCurrentRegion()
    ActiveCell.CurrentRegion.Select
End Sub
```

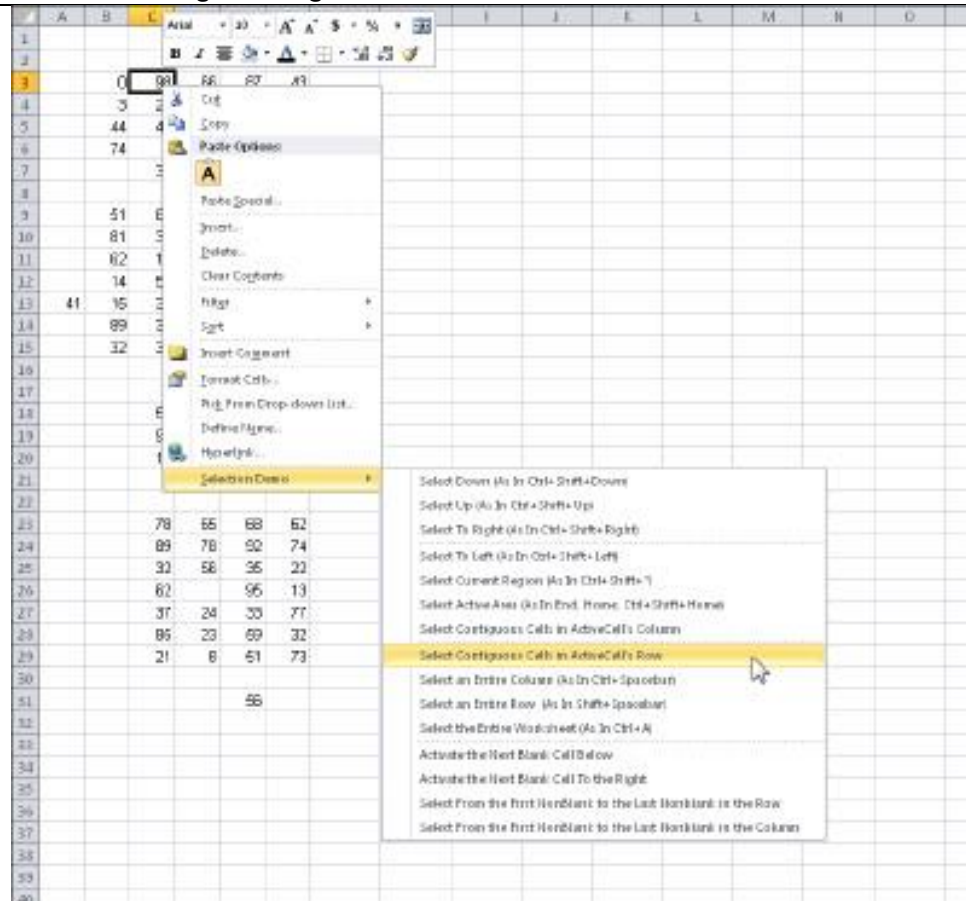


FIGURE 11-2: This workbook uses a custom shortcut menu to demonstrate how to select variably sized ranges by using VBA.

Often, you won't want to actually select the cells. Rather, you'll want to work with them in some way (for example, format them). You can easily adapt the cell-selecting procedures. The following procedure was adapted from `SelectCurrentRegion`. This procedure doesn't select cells; it applies formatting to the range that's defined as the current region around the active cell. You can also adapt the other procedures in the example workbook in this manner.

```
Sub FormatCurrentRegion()
    ActiveCell.CurrentRegion.Font.Bold = True
End Sub
```

### Prompting for a cell value

The following procedure demonstrates how to ask the user for a value and then insert it into cell A1 of the active worksheet:

```
Sub GetValue1()
    Range("A1").Value = InputBox("Enter the value")
End Sub
```



Figure 11-3 shows how the input box looks.



FIGURE 11-3: The InputBox function gets a value from the user to be inserted into a cell.

This procedure has a problem, however. If the user clicks the Cancel button in the input box, the procedure deletes any data already in the cell. The following modification takes no action if the Cancel button is clicked:

```
Sub GetValue2()
    Dim UserEntry As Variant
    UserEntry = InputBox("Enter the value")
    If UserEntry <> "" Then Range("A1").Value = UserEntry
End Sub
```

In many cases, you'll need to validate the user's entry in the input box. For example, you may require a number between 1 and 12. The following example demonstrates one way to validate the user's entry. In this example, an invalid entry is ignored, and the input box is displayed again. This cycle keeps repeating until the user enters a valid number or clicks Cancel.

```
Sub GetValue3()
    Dim UserEntry As Variant
    Dim Msg As String
    Const MinVal As Integer = 1
    Const MaxVal As Integer = 12
    Msg = "Enter a value between " & MinVal & " and " & MaxVal
    Do
        UserEntry = InputBox(Msg)
        If UserEntry = "" Then Exit Sub
        If IsNumeric(UserEntry) Then
            If UserEntry >= MinVal And UserEntry <= MaxVal Then Exit Do
        End If
        Msg = "Your previous entry was INVALID."
        Msg = Msg & vbCrLf
        Msg = Msg & "Enter a value between " & MinVal & " and " & MaxVal
    Loop
    ActiveSheet.Range("A1").Value = UserEntry
End Sub
```

As you can see in Figure 11-4, the code also changes the message displayed if the user makes an invalid entry.

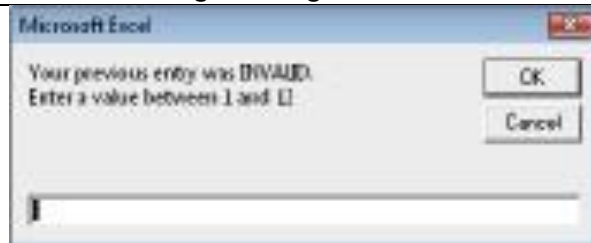


FIGURE 11-4: Validate a user's entry with the VBA InputBox function.

### Entering a value in the next empty cell

A common requirement is to enter a value into the next empty cell in a column or row. The following example prompts the user for a name and a value and then enters the data into the next empty row (see Figure 11-5).

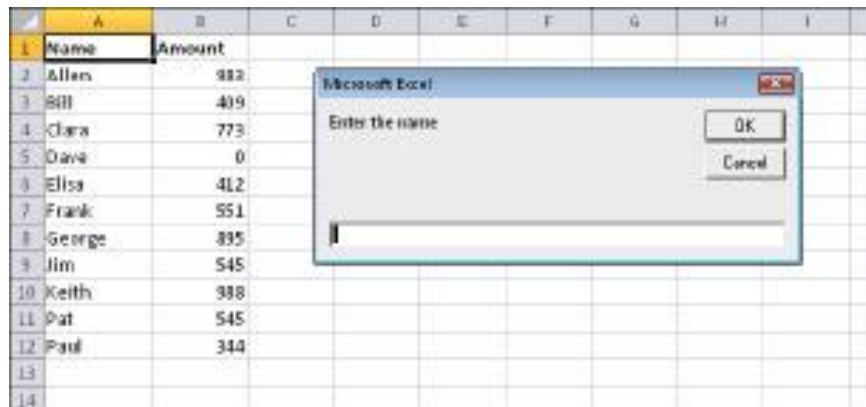


FIGURE 11-5: A macro for inserting data into the next empty row in a worksheet.

```

Sub GetData()
    Dim NextRow As Long
    Dim Entry1 As String, Entry2 As String
    Do
        'Determine next empty row
        NextRow = Cells(Rows.Count, 1).End(xlUp).Row + 1
        ' Prompt for the data
        Entry1 = InputBox("Enter the name")
        If Entry1 = "" Then Exit Sub
        Entry2 = InputBox("Enter the amount")
        If Entry2 = "" Then Exit Sub
        ' Write the data
        Cells(NextRow, 1) = Entry1
        Cells(NextRow, 2) = Entry2
    Loop
End Sub

```

To keep things simple, this procedure doesn't perform any validation. Notice that the loop continues indefinitely. I use Exit Sub statements to get out of the loop when the user clicks Cancel in the input box.

Notice the statement that determines the value of the NextRow variable. If you don't understand how this statement works, try the manual equivalent: Activate the last cell in column A (cell A1048576 in an Excel 2010 workbook), press End, and then press the up-arrow key. At this point, the last nonblank cell in column A will be selected. The Row property returns this row number, and it's incremented by 1 in order to get the row of the cell below it (the next empty row). Rather than hard-code the last cell in column A, I used RowsCount so that this procedure will work with previous versions of Excel (which have fewer rows).

Note that this technique of selecting the next empty cell has a slight glitch. If the column is completely empty, it will calculate row 2 as the next empty row. Writing additional code to account for this possibility would be fairly easy to do.

### Pausing a macro to get a user-selected range

In some situations, you may need an interactive macro. For example, you can create a macro that pauses while the user specifies a range of cells. The procedure in this section describes how to do this with Excel's InputBox method.

**Notes: Don't confuse Excel's InputBox method with VBA's InputBox function. Although these two items have the same name, they're not the same.**

The Sub procedure that follows demonstrates how to pause a macro and let the user select a range. The code then inserts a formula into each cell of the specified range.

```
Sub GetUserRange()  
    Dim UserRange As Range  
    Prompt = "Select a range for the random numbers."  
    Title = "Select a range"  
    ' Display the Input Box  
    On Error Resume Next  
    Set UserRange = Application.InputBox( _  
        Prompt:=Prompt, _  
        Title:=Title, _  
        Default:=ActiveCell.Address, _  
        Type:=8) 'Range selection  
    On Error GoTo 0  
    ' Was the Input Box canceled?  
    If UserRange Is Nothing Then  
        MsgBox "Canceled."  
    Else  
        UserRange.Formula = "=RAND()"  
    End If  
End Sub
```

The input box is shown in Figure 11-6.

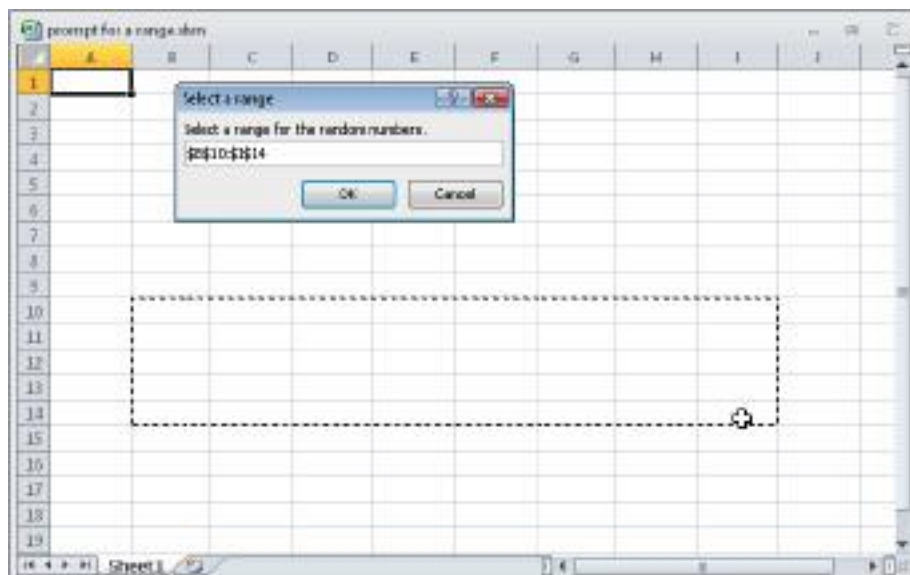


FIGURE 11-6: Use an input box to pause a macro.

Specifying a Type argument of 8 for the InputBox method is the key to this procedure. Also, note the use of On Error Resume Next. This statement ignores the error that occurs if the user clicks the Cancel button. If so, the UserRange object variable isn't defined. This example displays a message box with the text Canceled. If the user clicks OK, the macro continues. Using On Error GoTo 0 resumes normal error handling.

By the way, you don't need to check for a valid range selection. Excel takes care of this task for you.

**Notes:** Make sure that screen updating isn't turned off when you use the InputBox method to select a range. Otherwise, you won't be able to make a worksheet selection. Use the ScreenUpdating property of the Application object to control screen updating while a macro is running.

### Counting selected cells

You can create a macro that works with the range of cells selected by the user. Use the Count property of the Range object to determine how many cells are contained in a range selection (or any range, for that matter). For example, the following statement displays a message box that contains the number of cells in the current selection:

```
MsgBox Selection.Count
```

If the active sheet contains a range named data, the following statement assigns the number of cells in the data range to a variable named CellCount:

```
CellCount = Range("data").Count
```

You can also determine how many rows or columns are contained in a range. The following expression calculates the number of columns in the currently selected range:

```
Selection.Columns.Count
```

And, of course, you can also use the Rows property to determine the number of rows in a range. The following statement counts the number of rows in a range named data and assigns the number to a variable named RowCount:

---

```
RowCount = Range("data").Rows.Count
```

### Determining the type of selected range

Excel supports several types of range selections:

- A single cell
- A contiguous range of cells
- One or more entire columns
- One or more entire rows
- The entire worksheet
- Any combination of the preceding (that is, a multiple selection)

As a result, when your VBA procedure processes a user-selected range, you can't make any presumptions about what that range might be. For example, the range selection might consist of two areas, say A1:A10 and C1:C10. (To make a multiple selection, press Ctrl while you select the ranges with your mouse.)

In the case of a multiple range selection, the Range object comprises separate areas. To determine whether a selection is a multiple selection, use the Areas method, which returns an Areas collection. This collection represents all the ranges within a multiple range selection.

You can use an expression like the following to determine whether a selected range has multiple areas:

```
NumAreas = Selection.Areas.Count
```

If the NumAreas variable contains a value greater than 1, the selection is a multiple selection.

Following is a function named AreaType, which returns a text string that describes the type of range selection.

```
Function AreaType(RangeArea As Range) As String
```

```
    ' Returns the type of a range in an area
```

```
    Select Case True
```

```
        Case RangeArea.Cells.CountLarge = 1
```

```
            AreaType = "Cell"
```

```
        Case RangeArea.CountLarge = Cells.CountLarge
```

```
            AreaType = "Worksheet"
```

```
        Case RangeArea.Rows.Count = Cells.Rows.Count
```

```
            AreaType = "Column"
```

```
        Case RangeArea.Columns.Count = Cells.Columns.Count
```

```
            AreaType = "Row"
```

```
        Case Else
```

```
            AreaType = "Block"
```

```
    End Select
```

```
End Function
```

This function accepts a Range object as its argument and returns one of five strings that describe the area: Cell, Worksheet, Column, Row, or Block. The function uses a Select Case construct to determine which of five comparison expressions is True. For example, if the range consists of a single cell, the function returns Cell. If the number of cells in the range is equal to the number of

cells in the worksheet, it returns Worksheet. If the number of rows in the range equals the number of rows in the worksheet, it returns Column. If the number of columns in the range equals the number of columns in the worksheet, the function returns Row. If none of the Case expressions is True, the function returns Block.

Notice that I used the CountLarge property when counting cells. As I noted previously in this chapter, the number of selected cells could potentially exceed the limit of the Count property.

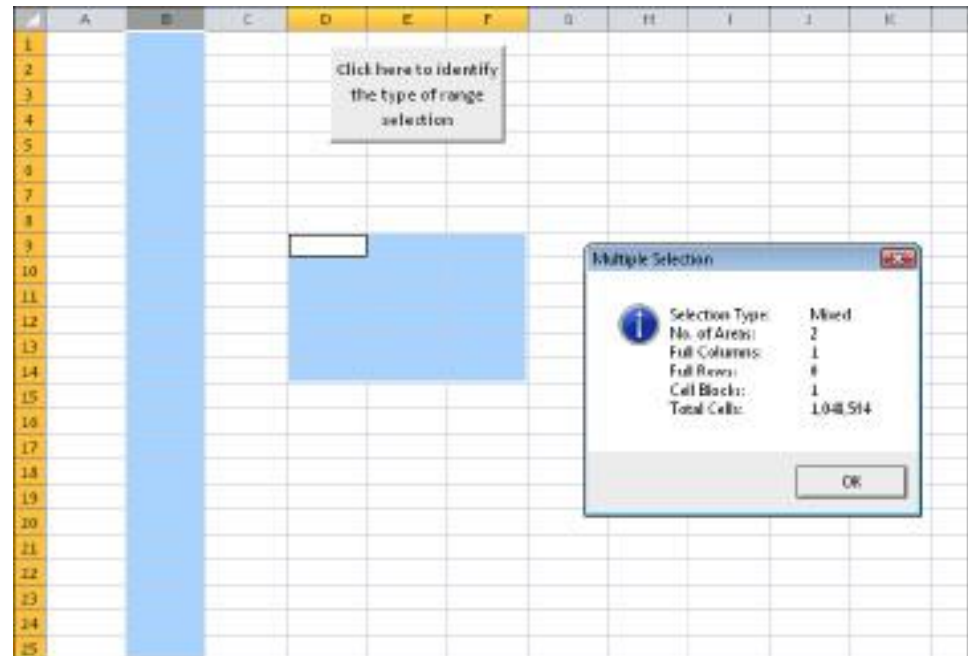


FIGURE 11-7: A VBA procedure analyzes the currently selected range.

### Looping through a selected range efficiently

A common task is to create a macro that evaluates each cell in a range and performs an operation if the cell meets a certain criterion. The procedure that follows is an example of such a macro. The ColorNegative procedure sets the cell's background color to red for cells that contain a negative value. For non-negative value cells, it sets the background color to none.

**Nots: This example is for educational purposes only. Using Excel's conditional formatting is a much better approach.**

```
Sub ColorNegative()  
    ' Makes negative cells red  
    Dim cell As Range  
    If TypeName(Selection) <> "Range" Then Exit Sub  
    Application.ScreenUpdating = False  
    For Each cell In Selection  
        If cell.Value < 0 Then  
            cell.Interior.Color = RGB(255, 0, 0)  
        Else  
            cell.Interior.Color = xlNone  
        End If  
    Next cell
```

End Sub

The ColorNegative procedure certainly works, but it has a serious flaw. For example, what if the used area on the worksheet were small, but the user selects an entire column? Or ten columns? Or the entire worksheet? You don't need to process all those empty cells, and the user would probably give up long before your code churns through all those cells.

A better solution (ColorNegative2) follows. In this revised procedure, I create a Range object variable, WorkRange, which consists of the intersection of the selected range and the worksheet's used range. Figure 11-8 shows an example; the entire column D is selected (1,048,576 cells). The worksheet's used range, however, consists of the range B2:I16. Therefore, the intersection of these ranges is D2:D16, which is a much smaller range than the original selection. The time difference between processing 15 cells versus processing 1,048,576 cells is significant.

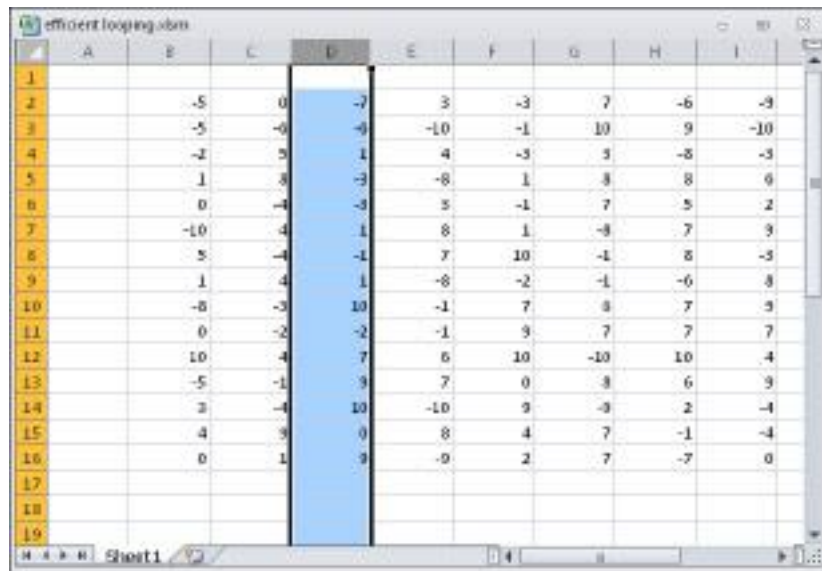


FIGURE 11-8: Using the intersection of the used range and the selected range results in fewer cells to process.

```
Sub ColorNegative2()
    ' Makes negative cells red
    Dim WorkRange As Range
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
    Application.ScreenUpdating = False
    Set WorkRange = Application.Intersect(Selection, _
    ActiveSheet.UsedRange)
    For Each cell In WorkRange
        If cell.Value < 0 Then
            cell.Interior.Color = RGB(255, 0, 0)
        Else
            cell.Interior.Color = xlNone
        End If
    Next cell
End Sub
```

The ColorNegative2 procedure is an improvement, but it's still not as efficient as it could be because it processes empty cells. A third revision, ColorNegative3, is quite a bit longer, but it's much more efficient. I use the SpecialCells method to generate two subsets of the selection: One subset (ConstantCells) includes only the cells with numeric constants; the other subset (FormulaCells) includes only the cells with numeric formulas. The code processes the cells in these subsets by using two For Each-Next constructs. The net effect: Only nonblank, nontext cells are evaluated, thus speeding up the macro considerably.

```
Sub ColorNegative3()  
    ' Makes negative cells red  
    Dim FormulaCells As Range, ConstantCells As Range  
    Dim cell As Range  
    If TypeName(Selection) <> "Range" Then Exit Sub  
    Application.ScreenUpdating = False  
    ' Create subsets of original selection  
    On Error Resume Next  
    Set FormulaCells = Selection.SpecialCells(xlFormulas, xlNumbers)  
    Set ConstantCells = Selection.SpecialCells(xlConstants, xlNumbers)  
    On Error GoTo 0  
    ' Process the formula cells  
    If Not FormulaCells Is Nothing Then  
        For Each cell In FormulaCells  
            If cell.Value < 0 Then  
                cell.Interior.Color = RGB(255, 0, 0)  
            Else  
                cell.Interior.Color = xlNone  
            End If  
        Next cell  
    End If  
    ' Process the constant cells  
    If Not ConstantCells Is Nothing Then  
        For Each cell In ConstantCells  
            If cell.Value < 0 Then  
                cell.Interior.Color = RGB(255, 0, 0)  
            Else  
                cell.Interior.Color = xlNone  
            End If  
        Next cell  
    End If  
End Sub
```

**Deleting all empty rows**



The following procedure deletes all empty rows in the active worksheet. This routine is fast and efficient because it doesn't check all rows. It checks only the rows in the used range, which is determined by using the UsedRange property of the Worksheet object.

```
Sub DeleteEmptyRows()  
    Dim LastRow As Long  
    Dim r As Long  
    Dim Counter As Long  
    Application.ScreenUpdating = False  
    LastRow = ActiveSheet.UsedRange.Rows.Count + _  
        ActiveSheet.UsedRange.Rows(1).Row - 1  
    For r = LastRow To 1 Step -1  
        If Application.WorksheetFunction.CountA(Rows(r)) = 0 Then  
            Rows(r).Delete  
            Counter = Counter + 1  
        End If  
    Next r  
    Application.ScreenUpdating = True  
    MsgBox Counter & " empty rows were deleted."  
End Sub
```

The first step is to determine the last used row and then assign this row number to the LastRow variable. This calculation isn't as simple as you might think because the used range may or may not begin in row 1. Therefore, LastRow is calculated by determining the number of rows in the used range, adding the first row number in the used range, and subtracting 1.

The procedure uses Excel's COUNTA worksheet function to determine whether a row is empty. If this function returns 0 for a particular row, the row is empty. Notice that the procedure works on the rows from bottom to top and also uses a negative step value in the For-Next loop. This negative step value is necessary because deleting rows causes all subsequent rows to move up in the worksheet. If the looping occurred from top to bottom, the counter within the loop wouldn't be accurate after a row is deleted.

The macro uses another variable, Counter, to keep track of how many rows were deleted. This number is displayed in a message box when the procedure ends.

### Duplicating rows a variable number of times

The example in this section demonstrates how to use VBA to create duplicates of a row. Figure 11-9 shows a worksheet for an office raffle. Column A contains the name, and column B contains the number of tickets purchased by each person. Column C contains a random number (generated by the RAND function). The winner will be determined by sorting the data based on column 3 (the highest random number wins).

	A	B	C	D
1	Name	No. Tickets	Random	
2	Alan	1	0.07987701	
3	Barbara	2	0.99031952	
4	Charlie	1	0.0094504	
5	Dave	5	0.90840464	
6	Frank	3	0.31514996	
7	Gilda	1	0.90027277	
8	Hubert	1	8.1048423	
9	Ivet	2	8.0910341	
10	Mark	1	0.50256856	
11	Noah	10	0.97290029	
12	Penelope	2	0.14935537	
13	Rance	1	0.05694515	
14	Wendy	2	0.30601043	
15				

FIGURE 11-9: The goal is to duplicate rows based on the value in column B.

The goal is to duplicate the rows so that each person will have a row for each ticket purchased. For example, Barbara purchased two tickets, so she should have two rows. The procedure to insert the new rows is shown here:

```

Sub DupeRows()
    Dim cell As Range
    ' 1st cell with number of tickets
    Set cell = Range("B2")
    Do While Not IsEmpty(cell)
        If cell > 1 Then
            Range(cell.Offset(1, 0), cell.Offset(cell.Value - 1, _
            0)).EntireRow.Insert
            Range(cell, cell.Offset(cell.Value - 1, 1)).EntireRow.FillDown
        End If
        Set cell = cell.Offset(cell.Value, 0)
    Loop
End Sub

```

The cell object variable is initialized to cell B2, the first cell that has a number. The loop inserts new rows and then copies the row using the FillDown method. The cell variable is incremented to the next person, and the loop continues until an empty cell is encountered. Figure 11-10 shows the worksheet after running this procedure.

	A	B	C	D
1	Name	No. Tickets	Random	
2	Alan	1	0.98771157	
3	Barbara	2	0.74151011	
4	Barbara	2	0.85528703	
5	Charlie	1	0.73332979	
6	Dave	5	0.99957885	
7	Dave	5	0.71642623	
8	Dave	5	0.32573479	
9	Dave	5	0.95313647	
10	Dave	5	0.09885253	
11	Frank	3	0.38698813	
12	Frank	3	0.48951989	
13	Frank	3	0.01047688	
14	Gilda	1	0.73986656	
15	Hubert	1	0.31302038	
16	Inez	2	0.09629546	
17	Inez	2	0.7408352	
18	Mark	1	0.18237242	
19	Norah	10	0.61515607	
20	Norah	10	0.37511779	
21	Norah	10	0.3017092	
22	Norah	10	0.03892754	
23	Norah	10	0.88545178	
24	Norah	10	0.38386286	
25	Norah	10	0.2008295	
26	Norah	10	0.57932413	
27	Norah	10	0.04228003	
28	Norah	10	0.47085364	
29	Penelope	2	0.78538233	
30	Penelope	2	0.88008608	
31	Rance	1	0.87725432	
32	Wendy	2	0.55074691	
33	Wendy	2	0.4916796	

FIGURE 11-10: New rows were added, according to the value in column B.

**Determining whether a range is contained in another range**

The following InRange function accepts two arguments, both Range objects. The function returns True if the first range is contained in the second range.

*Function InRange(rng1, rng2) As Boolean*

*' Returns True if rng1 is a subset of rng2*

*InRange = False*

*If rng1.Parent.Parent.Name = rng2.Parent.Parent.Name Then*

*If rng1.Parent.Name = rng2.Parent.Name Then*

*If Union(rng1, rng2).Address = rng2.Address Then*

*InRange = True*

*End If*

*End If*

*End If*

*End Function*

The InRange function may appear a bit more complex than it needs to be because the code needs to ensure that the two ranges are in the same worksheet and workbook. Notice that the procedure uses the Parent property, which returns an object's container object. For example, the following expression returns the name of the worksheet for the rng1 object reference:

*rng1.Parent.Name*

The following expression returns the name of the workbook for rng1:

*rng1.Parent.Parent.Name*

VBA's Union function returns a Range object that represents the union of two Range objects. The union consists of all the cells from both ranges. If the address of the union of the two ranges is the same as the address of the second range, the first range is contained within the second range.

### Determining a cell's data type

Excel provides a number of built-in functions that can help determine the type of data contained in a cell. These include ISTEXT, ISLOGICAL, and ISERROR. In addition, VBA includes functions such as IsEmpty, IsDate, and IsNumeric.

The following function, named CellType, accepts a range argument and returns a string (Blank, Text, Logical, Error, Date, Time, or Number) that describes the data type of the upper-left cell in the range. You can use this function in a worksheet formula or from another VBA procedure.

```
Function CellType(Rng) As String
    ' Returns the cell type of the upper left
    ' cell in a range
    Dim TheCell As Range
    Set TheCell = Rng.Range("A1")
    Select Case True
        Case IsEmpty(TheCell)
            CellType = "Blank"
        Case Application.IsText(TheCell)
            CellType = "Text"
        Case Application.IsLogical(TheCell)
            CellType = "Logical"
        Case Application.IsErr(TheCell)
            CellType = "Error"
        Case IsDate(TheCell)
            CellType = "Date"
        Case InStr(1, TheCell.Text, ".:") <> 0
            CellType = "Time"
        Case IsNumeric(TheCell)
            CellType = "Number"
    End Select
End Function
```

Notice the use of the Set TheCell statement. The CellType function accepts a range argument of any size, but this statement causes it to operate on only the upper-left cell in the range (which is represented by the TheCell variable).

### Reading and writing ranges

Many VBA tasks involve transferring values either from an array to a range or from a range to an array. Excel reads from ranges much faster than it writes to ranges because the latter operation involves the calculation engine. The

WriteReadRange procedure that follows demonstrates the relative speeds of writing and reading a range.

This procedure creates an array and then uses For-Next loops to write the array to a range and then read the range back into the array. It calculates the time required for each operation by using the Excel Timer function.

```
Sub WriteReadRange()  
    Dim MyArray()  
    Dim Time1 As Double  
    Dim NumElements As Long, i As Long  
    Dim WriteTime As String, ReadTime As String  
    Dim Msg As String  
    NumElements = 60000  
    ReDim MyArray(1 To NumElements)  
    ' Fill the array  
    For i = 1 To NumElements  
        MyArray(i) = i  
    Next i  
    ' Write the array to a range  
    Time1 = Timer  
    For i = 1 To NumElements  
        Cells(i, 1) = MyArray(i)  
    Next i  
    WriteTime = Format(Timer - Time1, "00:00")  
    ' Read the range into the array  
    Time1 = Timer  
    For i = 1 To NumElements  
        MyArray(i) = Cells(i, 1)  
    Next i  
    ReadTime = Format(Timer - Time1, "00:00")  
    ' Show results  
    Msg = "Write: " & WriteTime  
    Msg = Msg & vbCrLf  
    Msg = Msg & "Read: " & ReadTime  
    MsgBox Msg, vbOKOnly, NumElements & " Elements"  
End Sub
```

On my system, it took 58 seconds to write a 60,000-element array to a range, but it took less than 1 second to read the range into an array.

#### **A better way to write to a range**

The example in the preceding section uses a For-Next loop to transfer the contents of an array to a worksheet range. In this section, I demonstrate a more efficient way to accomplish this.

Start with the example that follows, which illustrates the most obvious (but not the most efficient) way to fill a range. This example uses a For-Next loop to insert its values in a range.

```
Sub LoopFillRange()  
    ' Fill a range by looping through cells  
    Dim CellsDown As Long, CellsAcross As Integer  
    Dim CurrRow As Long, CurrCol As Integer  
    Dim StartTime As Double  
    Dim CurrVal As Long  
    ' Get the dimensions  
    CellsDown = InputBox("How many cells down?")  
    If CellsDown = 0 Then Exit Sub  
    CellsAcross = InputBox("How many cells across?")  
    If CellsAcross = 0 Then Exit Sub  
    ' Record starting time  
    StartTime = Timer  
    ' Loop through cells and insert values  
    CurrVal = 1  
    Application.ScreenUpdating = False  
    For CurrRow = 1 To CellsDown  
        For CurrCol = 1 To CellsAcross  
            ActiveCell.Offset(CurrRow - 1, _  
                CurrCol - 1).Value = CurrVal  
            CurrVal = CurrVal + 1  
        Next CurrCol  
    Next CurrRow  
    ' Display elapsed time  
    Application.ScreenUpdating = True  
    MsgBox Format(Timer - StartTime, "00.00") & " seconds"  
End Sub
```

The example that follows demonstrates a much faster way to produce the same result. This code inserts the values into an array and then uses a single statement to transfer the contents of an array to the range.

```
Sub ArrayFillRange()  
    ' Fill a range by transferring an array  
    Dim CellsDown As Long, CellsAcross As Integer  
    Dim i As Long, j As Integer  
    Dim StartTime As Double  
    Dim TempArray() As Long  
    Dim TheRange As Range  
    Dim CurrVal As Long  
    ' Get the dimensions
```

```

CellsDown = InputBox("How many cells down?")
If CellsDown = 0 Then Exit Sub
CellsAcross = InputBox("How many cells across?")
If CellsAcross = 0 Then Exit Sub
' Record starting time
StartTime = Timer
' Redimension temporary array
ReDim TempArray(1 To CellsDown, 1 To CellsAcross)
' Set worksheet range
Set TheRange = ActiveCell.Range(Cells(1, 1), _
Cells(CellsDown, CellsAcross))
' Fill the temporary array
CurrVal = 0
Application.ScreenUpdating = False
For i = 1 To CellsDown
For j = 1 To CellsAcross
TempArray(i, j) = CurrVal + 1
CurrVal = CurrVal + 1
Next j
Next i
' Transfer temporary array to worksheet
TheRange.Value = TempArray
' Display elapsed time
Application.ScreenUpdating = True
MsgBox Format(Timer - StartTime, "00.00") & " seconds"
End Sub

```

On my system, using the loop method to fill a 1000 x 250–cell range (250,000 cells) took 10.05 seconds. The array transfer method took only 00.18 seconds to generate the same results — more than 50 times faster! The moral of this story? If you need to transfer large amounts of data to a worksheet, avoid looping whenever possible.

### Transferring one-dimensional arrays

The example in the preceding section involves a two-dimensional array, which works out nicely for row-and-column-based worksheets.

When transferring a one-dimensional array to a range, the range must be horizontal — that is, one row with multiple columns. If you need the data in a vertical range instead, you must first transpose the array to make it vertical. You can use Excel's TRANSPOSE function to do this. The following example transfers a 100-element array to a vertical worksheet range (A1:A100):

```

Range("A1:A100").Value =
Application.WorksheetFunction.Transpose(MyArray)

```

**Transferring a range to a variant array**

This section discusses yet another way to work with worksheet data in VBA. The following example transfers a range of cells to a two-dimensional variant array. Then message boxes display the upper bounds for each dimension of the variant array.

```
Sub RangeToVariant()  
    Dim x As Variant  
    x = Range("A1:L600").Value  
    MsgBox UBound(x, 1)  
    MsgBox UBound(x, 2)  
End Sub
```

In this example, the first message box displays 600 (the number of rows in the original range), and the second message box displays 12 (the number of columns). You'll find that transferring the range data to a variant array is virtually instantaneous.

The following example reads a range (named data) into a variant array, performs a simple multiplication operation on each element in the array, and then transfers the variant array back to the range:

```
Sub RangeToVariant2()  
    Dim x As Variant  
    Dim r As Long, c As Integer  
    ' Read the data into the variant  
    x = Range("data").Value  
    ' Loop through the variant array  
    For r = 1 To UBound(x, 1)  
        For c = 1 To UBound(x, 2)  
            ' Multiply by 2  
            x(r, c) = x(r, c) * 2  
        Next c  
    Next r  
    ' Transfer the variant back to the sheet  
    Range("data") = x  
End Sub
```

You'll find that this procedure runs amazingly fast. Working with 30,000 cells took less than one second.

**Selecting cells by value**

The example in this section demonstrates how to select cells based on their value. Oddly, Excel doesn't provide a direct way to perform this operation. My SelectByValue procedure follows. In this example, the code selects cells that contain a negative value, but you can easily change the code to select cells based on other criteria.

```
Sub SelectByValue()  
    Dim Cell As Object  
    Dim FoundCells As Range
```



```
Dim WorkRange As Range
If TypeName(Selection) <> "Range" Then Exit Sub
' Check all or selection?
If Selection.CountLarge = 1 Then
Set WorkRange = ActiveSheet.UsedRange
Else
Set WorkRange = Application.Intersect(Selection,
ActiveSheet.UsedRange)
End If
' Reduce the search to numeric cells only
On Error Resume Next
Set WorkRange = WorkRange.SpecialCells(xlConstants, xlNumbers)
If WorkRange Is Nothing Then Exit Sub
On Error GoTo 0
' Loop through each cell, add to the FoundCells range if it qualifies
For Each Cell In WorkRange
If Cell.Value < 0 Then
If FoundCells Is Nothing Then
Set FoundCells = Cell
Else
Set FoundCells = Union(FoundCells, Cell)
End If
End If
Next Cell
' Show message, or select the cells
If FoundCells Is Nothing Then
MsgBox "No cells qualify."
Else
FoundCells.Select
End If
End Sub
```

The procedure starts by checking the selection. If it's a single cell, then the entire worksheet is searched. If the selection is at least two cells, then only the selected range is searched. The range to be searched is further refined by using the SpecialCells method to create a Range object that consists only of the numeric constants.

The code within the For-Next loop examines the cell's value. If it meets the criterion (less than 0), then the cell is added to the FoundCells Range object by using the Union method. Note that you can't use the Union method for the first cell. If the FoundCells range contains no cells, attempting to use the Union method will generate an error. Therefore, the code checks whether FoundCells is Nothing.

When the loop ends, the FoundCells object will consist of the cells that meet the criterion (or will be Nothing if no cells were found). If no cells are found, a message box appears. Otherwise, the cells are selected.

### Copying a noncontiguous range

If you've ever attempted to copy a noncontiguous range selection, you discovered that Excel doesn't support such an operation. Attempting to do so brings up an error message: That command cannot be used on multiple selections.

An exception is when you attempt to copy a multiple selection that consists of entire rows or columns. Excel does allow that operation.

When you encounter a limitation in Excel, you can often circumvent it by creating a macro. The example in this section is a VBA procedure that allows you to copy a multiple selection to another location.

```
Sub CopyMultipleSelection()  
    Dim SelAreas() As Range  
    Dim PasteRange As Range  
    Dim UpperLeft As Range  
    Dim NumAreas As Long, i As Long  
    Dim TopRow As Long, LeftCol As Long  
    Dim RowOffset As Long, ColOffset As Long  
    If TypeName(Selection) <> "Range" Then Exit Sub  
    ' Store the areas as separate Range objects  
    NumAreas = Selection.Areas.Count  
    ReDim SelAreas(1 To NumAreas)  
    For i = 1 To NumAreas  
        Set SelAreas(i) = Selection.Areas(i)  
    Next  
    ' Determine the upper-left cell in the multiple selection  
    TopRow = ActiveSheet.Rows.Count  
    LeftCol = ActiveSheet.Columns.Count  
    For i = 1 To NumAreas  
        If SelAreas(i).Row < TopRow Then TopRow = SelAreas(i).Row  
        If SelAreas(i).Column < LeftCol Then LeftCol = SelAreas(i).Column  
    Next  
    Set UpperLeft = Cells(TopRow, LeftCol)  
    ' Get the paste address  
    On Error Resume Next  
    Set PasteRange = Application.InputBox _  
        (Prompt:="Specify the upper-left cell for the paste range:", _  
        Title:="Copy Multiple Selection", _  
        Type:=8)  
    On Error GoTo 0
```

```

' Exit if canceled
If TypeName(PasteRange) <> "Range" Then Exit Sub
' Make sure only the upper-left cell is used
Set PasteRange = PasteRange.Range("A1")
' Copy and paste each area
For i = 1 To NumAreas
RowOffset = SelAreas(i).Row - TopRow
ColOffset = SelAreas(i).Column - LeftCol
SelAreas(i).Copy PasteRange.Offset(RowOffset, ColOffset)
Next i
End Sub

```

Figure 11-11 shows the prompt to select the destination location.

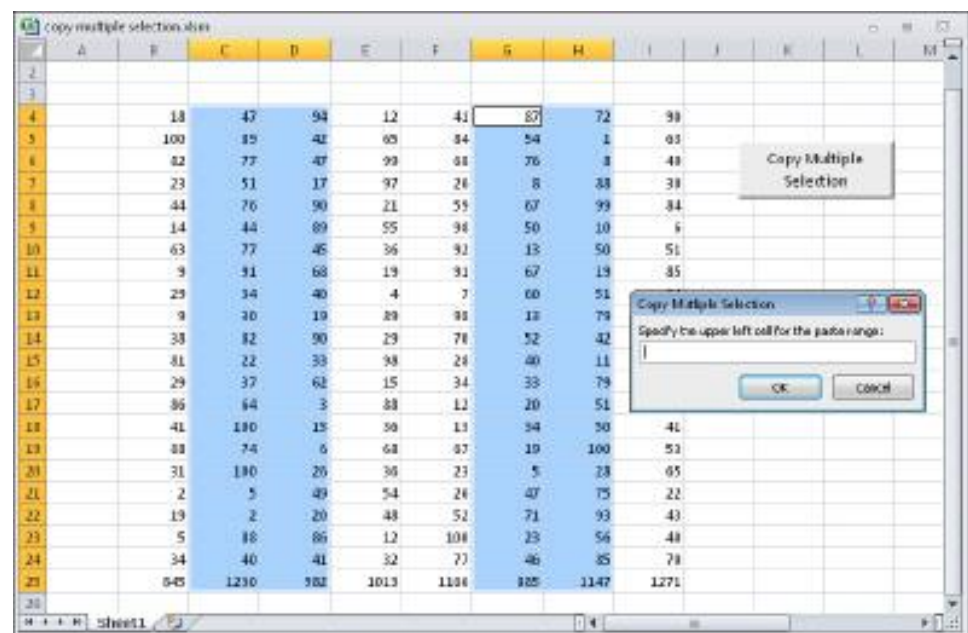


FIGURE 11-11: Using Excel's InputBox method to prompt for a cell location.

## 5.2. Working with Workbooks and Sheets

The examples in this section demonstrate various ways to use VBA to work with workbooks and worksheets.

### Saving all workbooks

The following procedure loops through all workbooks in the Workbooks collection and saves each file that has been saved previously:

```

Public Sub SaveAllWorkbooks()
Dim Book As Workbook
For Each Book In Workbooks
If Book.Path <> "" Then Book.Save
Next Book
End Sub

```

Notice the use of the Path property. If a workbook's Path property is empty, the file has never been saved (it's a newly created workbook). This procedure ignores such workbooks and saves only the workbooks that have a non-empty Path property.

### Saving and closing all workbooks

The following procedure loops through the Workbooks collection. The code saves and closes all workbooks.

```
Sub CloseAllWorkbooks()
    Dim Book As Workbook
    For Each Book In Workbooks
        If Book.Name <> ThisWorkbook.Name Then
            Book.Close savechanges:=True
        End If
    Next Book
    ThisWorkbook.Close savechanges:=True
End Sub
```

The procedure uses an If statement within the For-Next loop to determine whether the workbook is the workbook that contains the code. This statement is necessary because closing the workbook that contains the procedure would end the code, and subsequent workbooks wouldn't be affected. After all the other workbooks are closed, the workbook that contains the code closes itself.

### Hiding all but the selection

The example in this section hides all rows and columns in a worksheet except those in the current range selection. Figure 11-12 shows an example.

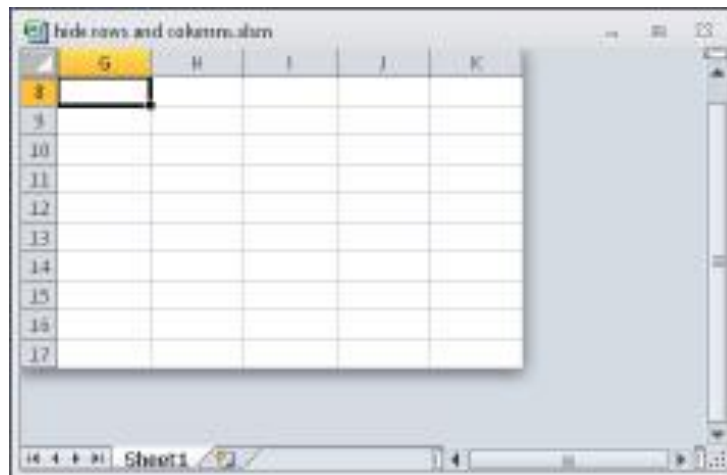


FIGURE 11-12: All rows and columns are hidden, except for a range (G8:K17).

```
Sub HideRowsAndColumns()
    Dim row1 As Long, row2 As Long
    Dim col1 As Long, col2 As Long
    If TypeName(Selection) <> "Range" Then Exit Sub
    ' If last row or last column is hidden, unhide all and quit
    If Rows(Rows.Count).EntireRow.Hidden Or _
```

```

Columns(Columns.Count).EntireColumn.Hidden Then
Cells.EntireColumn.Hidden = False
Cells.EntireRow.Hidden = False
Exit Sub
End If
row1 = Selection.Rows(1).Row
row2 = row1 + Selection.Rows.Count - 1
col1 = Selection.Columns(1).Column
col2 = col1 + Selection.Columns.Count - 1
Application.ScreenUpdating = False
On Error Resume Next
' Hide rows
Range(Cells(1, 1), Cells(row1 - 1, 1)).EntireRow.Hidden = True
Range(Cells(row2 + 1, 1), Cells(Rows.Count, 1)).EntireRow.Hidden =
True
' Hide columns
Range(Cells(1, 1), Cells(1, col1 - 1)).EntireColumn.Hidden = True
Range(Cells(1, col2 + 1), Cells(1,
Columns.Count)).EntireColumn.Hidden = True
End Sub

```

If the range selection consists of a noncontiguous range, the first area is used as the basis for hiding rows and columns.

### Synchronizing worksheets

If you use multisheet workbooks, you probably know that Excel can't synchronize the sheets in a workbook. In other words, there is no automatic way to force all sheets to have the same selected range and upper-left cell. The VBA macro that follows uses the active worksheet as a base and then performs the following on all other worksheets in the workbook:

- Selects the same range as the active sheet.
- Makes the upper-left cell the same as the active sheet.

Following is the listing for the procedure:

```

Sub SynchSheets()
' Duplicates the active sheet's active cell and upper left cell
' Across all worksheets
If TypeName(ActiveSheet) <> "Worksheet" Then Exit Sub
Dim UserSheet As Worksheet, sht As Worksheet
Dim TopRow As Long, LeftCol As Integer
Dim UserSel As String
Application.ScreenUpdating = False
' Remember the current sheet
Set UserSheet = ActiveSheet

```

```

' Store info from the active sheet
TopRow = ActiveWindow.ScrollRow
LeftCol = ActiveWindow.ScrollColumn
UserSel = ActiveWindow.RangeSelection.Address
' Loop through the worksheets
For Each sht In ActiveWorkbook.Worksheets
If sht.Visible Then 'skip hidden sheets
sht.Activate
Range(UserSel).Select
ActiveWindow.ScrollRow = TopRow
ActiveWindow.ScrollColumn = LeftCol
End If
Next sht
' Restore the original position
UserSheet.Activate
Application.ScreenUpdating = True
End Sub

```

### 5.3. VBA Techniques

The examples in this section illustrate common VBA techniques that you might be able to adapt to your own projects.

#### Toggling a Boolean property

A Boolean property is one that is either True or False. The easiest way to toggle a Boolean property is to use the Not operator, as shown in the following example, which toggles the WrapText property of a selection.

```

Sub ToggleWrapText()
' Toggles text wrap alignment for selected cells
If TypeName(Selection) = "Range" Then
Selection.WrapText = Not ActiveCell.WrapText
End If
End Sub

```

You can modify this procedure to toggle other Boolean properties.

Note that the active cell is used as the basis for toggling. When a range is selected and the property values in the cells are inconsistent (for example, some cells are bold, and others are not), it's considered mixed, and Excel uses the active cell to determine how to toggle. If the active cell is bold, for example, all cells in the selection are made not bold when you click the Bold button. This simple procedure mimics the way Excel works, which is usually the best practice.

Note also that this procedure uses the TypeName function to check whether the selection is a range. If the selection isn't a range, nothing happens.

You can use the Not operator to toggle many other properties. For example, to toggle the display of row and column borders in a worksheet, use the following code:

```

ActiveWindow.DisplayHeadings = Not ActiveWindow.DisplayHeadings

```

To toggle the display of gridlines in the active worksheet, use the following code:

```
ActiveWindow.DisplayGridlines = Not ActiveWindow.DisplayGridlines
```

### Determining the number of printed pages

If you need to determine the number of printed pages for a worksheet printout, you can use Excel's Print Preview feature and view the page count displayed at the bottom of the screen. The VBA procedure that follows calculates the number of printed pages for the active sheet by counting the number of horizontal and vertical page breaks:

```
Sub PageCount()  
    MsgBox (ActiveSheet.HPageBreaks.Count + 1) * _  
    (ActiveSheet.VPageBreaks.Count + 1) & " pages"  
End Sub
```

The following VBA procedure loops through all worksheets in the active workbook and displays the total number of printed pages, as shown in Figure 11-13:



FIGURE 11-13: Using VBA to count the number of printed pages in a workbook.

```
Sub ShowPageCount()  
    Dim PageCount As Integer  
    Dim sht As Worksheet  
    PageCount = 0  
    For Each sht In Worksheets  
        PageCount = PageCount + (sht.HPageBreaks.Count + 1) * _  
        (sht.VPageBreaks.Count + 1)  
    Next sht  
    MsgBox "Total printed pages = " & PageCount  
End Sub
```

### Displaying the date and time

If you understand the serial number system that Excel uses to store dates and times, you won't have any problems using dates and times in your VBA procedures.

The DateAndTime procedure displays a message box with the current date and time, as depicted in Figure 11-14. This example also displays a personalized message in the message box title bar.



FIGURE 11-14: A message box displaying the date and time.

The procedure uses the Date function as an argument for the Format function. The result is a string with a nicely formatted date. I used the same technique to get a nicely formatted time.

```

Sub DateAndTime()
    Dim TheDate As String, TheTime As String
    Dim Greeting As String
    Dim FullName As String, FirstName As String
    Dim SpaceInName As Long
    TheDate = Format(Date, "Long Date")
    TheTime = Format(Time, "Medium Time")
    ' Determine greeting based on time
    Select Case Time
        Case Is < TimeValue("12:00"): Greeting = "Good Morning, "
        Case Is >= TimeValue("17:00"): Greeting = "Good Evening, "
        Case Else: Greeting = "Good Afternoon, "
    End Select
    ' Append user's first name to greeting
    FullName = Application.UserName
    SpaceInName = InStr(1, FullName, " ", 1)
    ' Handle situation when name has no space
    If SpaceInName = 0 Then SpaceInName = Len(FullName)
    FirstName = Left(FullName, SpaceInName)
    Greeting = Greeting & FirstName
    ' Show the message
    MsgBox TheDate & vbCrLf & vbCrLf & "It's " & TheTime, vbOKOnly,
Greeting
End Sub

```

In the preceding example, I used named formats (Long Date and Medium Time) to ensure that the macro will work properly regardless of the user's international settings. You can, however, use other formats. For example, to display the date in mm/dd/yy format, you can use a statement like the following:

```
TheDate = Format(Date, "mm/dd/yy")
```

I used a Select Case construct to base the greeting displayed in the message box's title bar on the time of day. VBA time values work just as they do in Excel. If the time is less than .5 (noon), it's morning. If it's greater than .7083 (5 p.m.), it's evening. Otherwise, it's afternoon. I took the easy way out and used VBA's TimeValue function, which returns a time value from a string.



The next series of statements determines the user's first name, as recorded in the General tab in Excel's Options dialog box. I used VBA's InStr function to locate the first space in the user's name. When I first wrote this procedure, I didn't consider a username that has no space. So when I ran this procedure on a machine with a username of Nobody, the code failed — which goes to show you that I can't think of everything, and even the simplest procedures can run aground. (By the way, if the user's name is left blank, Excel always substitutes the name User.) The solution to this problem was to use the length of the full name for the SpaceInName variable so that the Left function extracts the full name.

The MsgBox function concatenates the date and time but uses the built-in vbCrLf constant to insert a line break between them. vbOKOnly is a predefined constant that returns 0, causing the message box to appear with only an OK button. The final argument is the Greeting, constructed earlier in the procedure.

### Getting a list of fonts

If you need to get a list of all installed fonts, you'll find that Excel doesn't provide a direct way to retrieve that information. The technique described here takes advantage of the fact that (for compatibility purposes) Excel 2010 still supports the old CommandBar properties and methods. These properties and methods were used in pre-Excel 2007 versions to work with toolbars and menus.

The ShowInstalledFonts macro displays a list of the installed fonts in column A of the active worksheet. It creates a temporary toolbar (a CommandBar object), adds the Font control, and reads the font names from that control. The temporary toolbar is then deleted.

```
Sub ShowInstalledFonts()  
    Dim FontList As CommandBarControl  
    Dim TempBar As CommandBar  
    Dim i As Long  
    ' Create temporary CommandBar  
    Set TempBar = Application.CommandBars.Add  
    Set FontList = TempBar.Controls.Add(ID:=1728)  
    ' Put the fonts into column A  
    Range("A:A").ClearContents  
    For i = 0 To FontList.ListCount - 1  
        Cells(i + 1, 1) = FontList.List(i + 1)  
    Next i  
    ' Delete temporary CommandBar  
    TempBar.Delete  
End Sub
```

**Notes:** As an option, you can display each font name in the actual font (as shown in Figure 11-15). To do so, add this statement inside the For-Next loop:

```
Cells(i+1,1).Font.Name = FontList.List(i+1)
```

**Be aware, however, that using many fonts in a workbook can eat up lots of system resources, and it could even crash your system.**



FIGURE 11-15: Listing font names in the actual fonts.

### Sorting an array

Although Excel has a built-in command to sort worksheet ranges, VBA doesn't offer a method to sort arrays. One viable (but cumbersome) workaround is to transfer your array to a worksheet range, sort it by using Excel's commands, and then return the result to your array. But if speed is essential, it's better to write a sorting routine in VBA.

In this section, I cover four different sorting techniques:

- Worksheet sort transfers an array to a worksheet range, sorts it, and transfers it back to the array. This procedure accepts an array as its only argument.
- Bubble sort is a simple sorting technique (also used in the Chapter 9 sheet-sorting example). Although easy to program, the bubble-sorting algorithm tends to be rather slow, especially when the number of elements is large.
- Quick sort is a much faster sorting routine than bubble sort, but it is also more difficult to understand. This technique works only with Integer and Long data types.
- Counting sort is lightning fast, but also difficult to understand. Like the quick sort, this technique works only with Integer and Long data types.

Figure 11-16 shows the dialog box for this project. I tested the sorting procedures with seven different array sizes, ranging from 100 to 100,000 elements. The arrays contained random numbers (of type Long).



FIGURE 11-16: Comparing the time required to perform sorts of various array sizes.

Table 11-1 shows the results of my tests. A 0.00 entry means that the sort was virtually instantaneous (less than .01 second).

**Table 11-1: Sorting Times (in Seconds) for Four Sort Algorithms Using Randomly Filled Arrays**

Array Elements	Excel Worksheet Sort	VBA Bubble Sort	VBA Quick Sort	VBA Counting Sort
100	0.04	0.00	0.00	0.02
500	0.02	0.01	0.00	0.01
1,000	0.03	0.03	0.00	0.00
5,000	0.07	0.84	0.01	0.01
10,000	0.09	3.41	0.01	0.01
50,000	0.43	79.95	0.07	0.02
100,000	0.78	301.90	0.14	0.04

The worksheet sort algorithm is amazingly fast, especially when you consider that the values are transferred to the sheet, sorted, and then transferred back to the array.

The bubble sort algorithm is reasonably fast with small arrays, but for larger arrays (more than 10,000 elements), forget it. The quick sort and counting sort algorithms are blazingly fast, but they're limited to Integer and Long data types.

### Processing a series of files

One common use for macros is to perform repetitive tasks. The example in this section demonstrates how to execute a macro on several different files stored on disk. This example — which may help you set up your own routine for this type of task — prompts the user for a file specification and then processes all matching files. In this case, processing consists of importing the file and entering a series of summary formulas that describe the data in the file.

```
Sub BatchProcess()
    Dim FileSpec As String
    Dim i As Integer
    Dim FileName As String
```

```

Dim FileList() As String
Dim FoundFiles As Integer
' Specify path and file spec
FileSpec = ThisWorkbook.Path & "\ " & "text???.txt"
FileName = Dir(FileSpec)
' Was a file found?
If FileName <> "" Then
    FoundFiles = 1
    ReDim Preserve FileList(1 To FoundFiles)
    FileList(FoundFiles) = FileName
Else
    MsgBox "No files were found that match " & FileSpec
Exit Sub
End If
' Get other filenames
Do
    FileName = Dir
    If FileName = "" Then Exit Do
    FoundFiles = FoundFiles + 1
    ReDim Preserve FileList(1 To FoundFiles)
    FileList(FoundFiles) = FileName & "*"
Loop
' Loop through the files and process them
For i = 1 To FoundFiles
    Call ProcessFiles(FileList(i))
Next i
End Sub

```

The matching filenames are stored in an array named FoundFiles, and the procedure uses a For-Next loop to process the files. Within the loop, the processing is done by calling the ProcessFiles procedure, which follows. This simple procedure uses the OpenText method to import the file and then inserts five formulas. You may, of course, substitute your own routine in place of this one:

```

Sub ProcessFiles(FileName As String)
' Import the file
Workbooks.OpenText FileName:=FileName, _
    Origin:=xlWindows, _
    StartRow:=1, _
    DataType:=xlFixedWidth, _
    FieldInfo:= _
    Array(Array(0, 1), Array(3, 1), Array(12, 1))

```

#### 5.4. Some Useful Functions for Use in Your Code

*' Enter summary formulas*

*Range("D1").Value = "A"*

*Range("D2").Value = "B"*

*Range("D3").Value = "C"*

*Range("E1:E3").Formula = "=COUNTIF(B:B,D1)"*

*Range("F1:F3").Formula = "=SUMIF(B:B,D1,C:C)"*

*End Sub*

In this section, I present some custom utility functions that you may find useful in your own applications and that may provide inspiration for creating similar functions. These functions are most useful when called from another VBA procedure. Therefore, they're declared by using the Private keyword and thus won't appear in Excel's Insert Function dialog box.

##### The FileExists function

This function takes one argument (a path with filename) and returns True if the file exists:

*Private Function FileExists(fname) As Boolean*

*' Returns TRUE if the file exists*

*FileExists = (Dir(fname) <> "")*

*End Function*

##### The FileNameOnly function

This function accepts one argument (a path with filename) and returns only the filename. In other words, it strips out the path.

*Private Function FileNameOnly(pname) As String*

*' Returns the filename from a path/filename string*

*Dim temp As Variant*

*length = Len(pname)*

*temp = Split(pname, Application.PathSeparator)*

*FileNameOnly = temp(UBound(temp))*

*End Function*

The function uses the VBA Split function, which accepts a string (that includes delimiter characters), and returns a variant array that contains the elements between the delimiter characters. In this case the temp variable contains an array that consists of each text string between the Application.PathSeparator (usually a backslash character). For another example of the Split function, see "Extracting the nth element from a string," later in this chapter.

If the argument is c:\excel files\2010\backup\budget.xlsx, the function returns the string budget.xlsx.

The FileNameOnly function works with any path and filename (even if the file does not exist). If the file exists, the following function is a simpler way to strip off the path and return only the filename:

*Private Function FileNameOnly2(pname) As String*

```
FileNameOnly2 = Dir(pname)
End Function
```

### The PathExists function

This function accepts one argument (a path) and returns True if the path exists:

```
Private Function PathExists(pname) As Boolean
    ' Returns TRUE if the path exists
    If Dir(pname, vbDirectory) = "" Then
        PathExists = False
    Else
        PathExists = (GetAttr(pname) And vbDirectory) = vbDirectory
    End If
End Function
```

### The RangeNameExists function

This function accepts a single argument (a range name) and returns True if the range name exists in the active workbook:

```
Private Function RangeNameExists(nname) As Boolean
    ' Returns TRUE if the range name exists
    Dim n As Name
    RangeNameExists = False
    For Each n In ActiveWorkbook.Names
        If UCase(n.Name) = UCase(nname) Then
            RangeNameExists = True
        End If
    Next n
End Function
```

Another way to write this function follows. This version attempts to create an object variable using the name. If doing so generates an error, then the name doesn't exist.

```
Private Function RangeNameExists2(nname) As Boolean
    ' Returns TRUE if the range name exists
    Dim n As Range
    On Error Resume Next
    Set n = Range(nname)
    If Err.Number = 0 Then RangeNameExists2 = True _
    Else RangeNameExists2 = False
End Function
```

**Testing for membership in a collection**

The following function procedure is a generic function that you can use to determine whether an object is a member of a collection:

```
Private Function IsInCollection(Coln As Object, _  
    Item As String) As Boolean  
    Dim Obj As Object  
    On Error Resume Next  
    Set Obj = Coln(Item)  
    IsInCollection = Not Obj Is Nothing  
End Function
```

This function accepts two arguments: the collection (an object) and the item (a string) that might or might not be a member of the collection. The function attempts to create an object variable that represents the item in the collection. If the attempt is successful, the function returns True; otherwise, it returns False.

You can use the IsInCollection function in place of three other functions listed in this chapter: RangeNameExists, SheetExists, and WorkbooksOpen. To determine whether a range named Data exists in the active workbook, call the IsInCollection function with this statement:

```
MsgBox IsInCollection(ActiveWorkbook.Names, "Data")
```

To determine whether a workbook named Budget is open, use this statement:

```
MsgBox IsInCollection(Workbooks, "budget.xlsx")
```

To determine whether the active workbook contains a sheet named Sheet1, use this statement:

```
MsgBox IsInCollection(ActiveWorkbook.Worksheets, "Sheet1")
```

**The SheetExists function**

This function accepts one argument (a worksheet name) and returns True if the worksheet exists in the active workbook:

```
Private Function SheetExists(sname) As Boolean  
    ' Returns TRUE if sheet exists in the active workbook  
    Dim x As Object  
    On Error Resume Next  
    Set x = ActiveWorkbook.Sheets(sname)  
    If Err.Number = 0 Then SheetExists = True _  
    Else SheetExists = False  
End Function
```

**The WorkbooksOpen function**

This function accepts one argument (a workbook name) and returns True if the workbook is open:

```
Private Function WorkbooksOpen(wbname) As Boolean  
    ' Returns TRUE if the workbook is open  
    Dim x As Workbook  
    On Error Resume Next
```

```

Set x = Workbooks(wbname)

If Err.Number = 0 Then WorkbookIsOpen = True _
Else WorkbookIsOpen = False

End Function

```

### Retrieving a value from a closed workbook

VBA doesn't include a method to retrieve a value from a closed workbook file. You can, however, take advantage of Excel's ability to work with linked files. This section contains a custom VBA function (GetValue, which follows) that retrieves a value from a closed workbook. It does so by calling an XLM macro, which is an old-style macro used in versions prior to Excel 5. Fortunately, Excel still supports this old macro system.

```

Private Function GetValue(path, file, sheet, ref)

' Retrieves a value from a closed workbook

Dim arg As String

' Make sure the file exists

If Right(path, 1) <> "\" Then path = path & "\"

If Dir(path & file) = "" Then

GetValue = "File Not Found"

Exit Function

End If

' Create the argument

arg = "" & path & "[" & file & "]" & sheet & "!" & _
Range(ref).Range("A1").Address(, , xLR1C1)

' Execute an XLM macro

GetValue = ExecuteExcel4Macro(arg)

End Function

```

The GetValue function takes four arguments:

- **path:** The drive and path to the closed file (for example, "d:\files")
- **file:** The workbook name (for example, "budget.xlsx")
- **sheet:** The worksheet name (for example, "Sheet1")
- **ref:** The cell reference (for example, "C4")

The following Sub procedure demonstrates how to use the GetValue function. It displays the value in cell A1 in Sheet1 of a file named 2010budget.xlsx, located in the XLFiles\Budget directory on drive C.

```

Sub TestGetValue()

Dim p As String, f As String

Dim s As String, a As String

p = "c:\XLFiles\Budget"

f = "2010budget.xlsx"

s = "Sheet1"

```



## 5.5. Some Useful Worksheet Functions

```
a = "A1"
```

```
MsgBox GetValue(p, f, s, a)
```

```
End Sub
```

Another example follows. This procedure reads 1,200 values (100 rows and 12 columns) from a closed file and then places the values into the active worksheet.

```
Sub TestGetValue2()
```

```
Dim p As String, f As String
```

```
Dim s As String, a As String
```

```
Dim r As Long, c As Long
```

```
p = "c:\XLFiles\Budget"
```

```
f = "2010Budget.xlsx"
```

```
s = "Sheet1"
```

```
Application.ScreenUpdating = False
```

```
For r = 1 To 100
```

```
For c = 1 To 12
```

```
a = Cells(r, c).Address
```

```
Cells(r, c) = GetValue(p, f, s, a)
```

```
Next c
```

```
Next r
```

```
End Sub
```

The examples in this section are custom functions that you can use in worksheet formulas. Remember, you must define these Function procedures in a VBA module (not a code module associated with ThisWorkbook, a Sheet, or a UserForm).

### Returning cell formatting information

This section contains a number of custom functions that return information about a cell's formatting. These functions are useful if you need to sort data based on formatting (for example, sort in such a way that all bold cells are together).

**Notes:** You'll find that these functions aren't always updated automatically. This is because changing formatting, for example, doesn't trigger Excel's recalculation engine. To force a global recalculation (and update all the custom functions), press **Ctrl+Alt+F9**.

**Alternatively, you can add the following statement to your function:**

```
Application.Volatile
```

**When this statement is present, then pressing F9 will recalculate the function.**

The following function returns TRUE if its single-cell argument has bold formatting. If a range is passed as the argument, the function uses the upper-left cell of the range.

```
Function IsBold(cell) As Boolean
```

```
    ' Returns TRUE if cell is bold
```

```
IsBold = cell.Range("A1").Font.Bold  
End Function
```

Note that this function works only with explicitly applied formatting. It doesn't work for formatting applied using conditional formatting. Excel 2010 introduced a new object, `DisplayFormat`. This object takes conditional formatting into account. Here's the `IsBold` function rewritten so that it also works with bold formatting applied as a result of conditional formatting:

```
Function IsBold(cell) As Boolean  
    ' Returns TRUE if cell is bold, even if from conditional formatting  
    IsBold = cell.Range("A1").DisplayFormat.Font.Bold  
End Function
```

The following function returns `TRUE` if its single-cell argument has italic formatting:

```
Function IsItalic(cell) As Boolean  
    ' Returns TRUE if cell is italic  
    IsItalic = cell.Range("A1").Font.Italic  
End Function
```

Both of the preceding functions will return an error if the cell has mixed formatting — for example, if only some characters are bold. The following function returns `TRUE` only if all characters in the cell are bold:

```
Function AllBold(cell) As Boolean  
    ' Returns TRUE if all characters in cell are bold  
    If IsNull(cell.Font.Bold) Then  
        AllBold = False  
    Else  
        AllBold = cell.Font.Bold  
    End If  
End Function
```

You can simplify the `AllBold` function as follows:

```
Function AllBold (cell) As Boolean  
    ' Returns TRUE if all characters in cell are bold  
    AllBold = Not IsNull(cell.Font.Bold)  
End Function
```

The `FillColor` function returns an integer that corresponds to the color index of the cell's interior. The actual color depends on the workbook theme that's applied. If the cell's interior isn't filled, the function returns `-4142`.

This function doesn't work with fill colors applied in tables (created with `Insert⇒Tables⇒Table`) or pivot tables. You need to use the `DisplayFormat` object to detect that type of fill color, as I described previously.

```
Function FillColor(cell) As Integer
    ' Returns an integer corresponding to
    ' cell's interior color
    FillColor = cell.Range("A1").Interior.ColorIndex
End Function
```

### A talking worksheet

The SayIt function uses Excel's text-to-speech generator to "speak" its argument (which can be literal text or a cell reference).

```
Function SayIt(txt)
    Application.Speech.Speak (txt)
    SayIt = txt
End Function
```

This function has some amusing possibilities, but it can also be useful. For example, use the function in a formula like this:

```
=IF(SUM(A:A)>25000,SayIt("Goal Reached"))
```

If the sum of the values in column A exceeds 25,000, you'll hear the synthesized voice tell you that the goal has been reached. You can also use the Speak method at the end of a lengthy procedure. That way, you can do something else, and you'll get an audible notice when the procedure ends.

### Displaying the date when a file was saved or printed

An Excel workbook contains several built-in document properties, accessible from the BuiltinDocumentProperties property of the Workbook object. The following function returns the date and time that the workbook was last saved:

```
Function LastSaved()
    Application.Volatile
    LastSaved = ThisWorkbook. _
        BuiltinDocumentProperties("Last Save Time")
End Function
```

The date and time returned by this function are the same date and time that appear in the Related Dates section of Backstage View when you choose File⇒Info. Note that the AutoSave feature also affects this value. In other words, the "Last Save Time" is not necessarily the last time the file was saved by the user.

The following function is similar to LastSaved, but it returns the date and time when the workbook was last printed or previewed. If the workbook has never been printed or previewed, the function returns a #VALUE error.

```
Function LastPrinted()
    Application.Volatile
    LastPrinted = ThisWorkbook. _
        BuiltinDocumentProperties("Last Print Date")
End Function
```

If you use these functions in a formula, you might need to force a recalculation (by pressing F9) to get the current values of these properties.

**Notes:** Quite a few additional built-in properties are available, but Excel doesn't use all of them. For example, attempting to access the Number of Bytes property will generate an error. For a list of all built-in properties, consult the Help system.

The preceding LastSaved and LastPrinted functions are designed to be stored in the workbook in which they're used. In some cases, you may want to store the function in a different workbook (for example, personal.xlsb) or in an add-in. Because these functions reference ThisWorkbook, they won't work correctly. Following are more general-purpose versions of these functions. These functions use [ApplicationCaller](#), which returns a Range object that represents the cell that calls the function. The use of Parent.Parent returns the workbook (that is, the parent of the parent of the Range object — a Workbook object). This topic is explained further in the next section.

```
Function LastSaved2()  
Application.Volatile  
LastSaved2 = Application.Caller.Parent.Parent. _  
BuiltInDocumentProperties("Last Save Time")  
End Function
```

### Understanding object parents

As you know, Excel's object model is a hierarchy: Objects are contained in other objects. At the top of the hierarchy is the Application object. Excel contains other objects, and these objects contain other objects, and so on. The following hierarchy depicts how a Range object fits into this scheme:

- Application object
- Workbook object
- Worksheet object
- Range object

In the lingo of object-oriented programming, a Range object's parent is the Worksheet object that contains it. A Worksheet object's parent is the Workbook object that contains the worksheet, and a Workbook object's parent is the Application object.

How can you put this information to use? Examine the SheetName VBA function that follows. This function accepts a single argument (a range) and returns the name of the worksheet that contains the range. It uses the Parent property of the Range object. The Parent property returns an object: the object that contains the Range object.

```
Function SheetName(ref) As String  
SheetName = ref.Parent.Name  
End Function
```

The next function, WorkbookName, returns the name of the workbook for a particular cell. Notice that it uses the Parent property twice. the first Parent property returns a Worksheet object, and the second Parent property returns a Workbook object.

```
Function WorkbookName(ref) As String  
WorkbookName = ref.Parent.Parent.Name  
End Function
```

The AppName function that follows carries this exercise to the next logical level, accessing the Parent property three times (the parent of the parent of the parent). This function returns the name of the Application object for a particular cell. It will, of course, always return Microsoft Excel.

```
Function AppName(ref) As String
    AppName = ref.Parent.Parent.Parent.Name
End Function
```

### Counting cells between two values

The following function, named CountBetween, returns the number of values in a range (first argument) that fall between values represented by the second and third arguments:

```
Function CountBetween(InRange, num1, num2) As Long
    ' Counts number of values between num1 and num2
    With Application.WorksheetFunction
        If num1 <= num2 Then
            CountBetween = .Countifs(InRange, ">=" & num1, _
                InRange, "<=" & num2)
        Else
            CountBetween = .Countifs(InRange, ">=" & num2, _
                InRange, "<=" & num1)
        End If
    End With
End Function
```

Note that this function uses Excel's COUNTIFS function. In fact, the CountBetween function is essentially a wrapper that can simplify your formulas.

Following is an example formula that uses the CountBetween function. The formula returns the number of cells in A1:A100 that are greater than or equal to 10 and less than or equal to 20.

```
=CountBetween(A1:A100,10,20)
```

The function accepts the two numeric argument in either order. So, this formula is equivalent to the previous formula:

```
=CountBetween(A1:A100,20,10)
```

Using this VBA function is simpler than entering the following (somewhat confusing) formula:

```
=COUNTIFS(A1:A100,">=10",A1:A100,"<=20")
```

### Determining the last non-empty cell in a column or row

In this section, I present two useful functions: LastInColumn returns the contents of the last non-empty cell in a column; LastInRow returns the contents of the last non-empty cell in a row. Each function accepts a range as its single argument. The range argument can be a complete column (for LastInColumn) or a complete row (for LastInRow). If the supplied argument isn't a complete column or row, the function uses the column or row of the upper-left cell in the range. For example, the following formula returns the last value in column B:

*=LastInColumn(B5)*

The following formula returns the last value in row 7:

*=LastInRow(C7:D9)*

The LastInColumn function follows:

*Function LastInColumn(rng As Range)*

*‘ Returns the contents of the last non-empty cell in a column*

*Dim LastCell As Range*

*Application.Volatile*

*With rng.Parent*

*With .Cells(.Rows.Count, rng.Column)*

*If Not IsEmpty(.Value) Then*

*LastInColumn = .Value*

*ElseIf IsEmpty(.End(xlUp)) Then*

*LastInColumn = ""*

*Else*

*LastInColumn = .End(xlUp).Value*

*End If*

*End With*

*End With*

*End Function*

This function is rather complicated, so here are a few points that may help you understand it:

- Application.Volatile causes the function to be executed whenever the sheet is calculated.
- RowsCount returns the number of rows in the worksheet. I used the Count property, rather than hard-coding the value, because not all worksheets have the same number of rows.
- rngColumn returns the column number of the upper-left cell in the rng argument.
- Using rng.Parent causes the function to work properly even if the rng argument refers to a different sheet or workbook.
- The End method (with the xlUp argument) is equivalent to activating the last cell in a column, pressing End, and then pressing the up-arrow key.
- The IsEmpty function checks whether the cell is empty. If so, it returns an empty string. Without this statement, an empty cell would be returned as 0.

The LastInRow function follows. This function is very similar to the LastInColumn function.

*Function LastInRow(rng As Range)*

*‘ Returns the contents of the last non-empty cell in a row*

*Application.Volatile*

*With rng.Parent*

*With .Cells(rng.Row, .Columns.Count)*

*If Not IsEmpty(.Value) Then*

```

LastInRow = .Value
ElseIf IsEmpty(.End(xlToLeft)) Then
LastInRow = ""
Else
LastInRow = .End(xlToLeft).Value
End If
End With
End With
End Function

```

### Does a string match a pattern?

The IsLike function is very simple (but also very useful). This function returns TRUE if a text string matches a specified pattern.

This function, which follows, is remarkably simple. As you can see, the function is essentially a wrapper that lets you take advantage of VBA's powerful Like operator in your formulas.

```

Function IsLike(text As String, pattern As String) As Boolean
' Returns true if the first argument is like the second
IsLike = text Like pattern
End Function

```

This IsLike function takes two arguments:

- **text:** A text string or a reference to a cell that contains a text string
- **pattern:** A string that contains wildcard characters according to the following list:

Character(s) in Pattern	Matches in Text
?	Any single character
*	Zero or more characters
#	Any single digit (0–9)
[charlist]	Any single character in charlist
[!charlist]	Any single character not in charlist

The following formula returns TRUE because \* matches any number of characters. It returns TRUE if the first argument is any text that begins with g.

```
=IsLike("guitar","g*")
```

The following formula returns TRUE because ? matches any single character. If the first argument were "Unit12", the function would return FALSE.

```
=IsLike("Unit1","Unit?")
```

The next formula returns TRUE because the first argument is a single character in the second argument.

```
=ISLIKE("a","[aeiou]")
```

The following formula returns TRUE if cell A1 contains a, e, i, o, u, A, E, I, O, or U. Using the UPPER function for the arguments makes the formula not case-sensitive.

`=IsLike(UPPER(A1), UPPER("[aeiou]"))`

The following formula returns TRUE if cell A1 contains a value that begins with 1 and has exactly three digits (that is, any integer between 100 and 199).

`=IsLike(A1, "1##")`

### Extracting the nth element from a string

ExtractElement is a custom worksheet function (which you can also call from a VBA procedure) that extracts an element from a text string. For example, if a cell contains the following text, you can use the ExtractElement function to extract any of the substrings between the hyphens.

123-456-789-0133-8844

The following formula, for example, returns 0133, which is the fourth element in the string. The string uses a hyphen (-) as the separator.

`=ExtractElement("123-456-789-0133-8844",4,"-")`

The ExtractElement function uses three arguments:

- Txt: The text string from which you're extracting. It can be a literal string or a cell reference.
- n: An integer that represents the element to extract.
- Separator: A single character used as the separator.

**Notes: If you specify a space as the Separator character, multiple spaces are treated as a single space, which is almost always what you want. If n exceeds the number of elements in the string, the function returns an empty string.**

The VBA code for the ExtractElement function follows:

```
Function ExtractElement(Txt, n, Separator) As String
    ' Returns the nth element of a text string, where the
    ' elements are separated by a specified separator character
    Dim AllElements As Variant
    AllElements = Split(Txt, Separator)
    ExtractElement = AllElements(n - 1)
End Function
```

This function uses VBA's Split function, which returns a variant array that contains each element of the text string. This array begins with 0 (not 1), so using n - 1 references the desired element.

### Spelling out a number

The SPELLDOLLARS function returns a number spelled out in text — as on a check. For example, the following formula returns the string One hundred twenty-three and 45/100 dollars:

`=SPELLDOLLARS(123.45)`



Figure 11-17 shows some additional examples of the SPELLDOLLARS function. Column C contains formulas that use the function. For example, the formula in C1 is

`=SPELLDOLLARS(A1)`

Note that negative numbers are spelled out and enclosed in parentheses.

	A	B	C	D	E	F	G	H	I	J
1	12		Thirty-Two and 00/100 Dollars							
2	37.95		Thirty-Seven and 95/100 Dollars							
3	-12		(Thirty-Two and 00/100 Dollars)							
4	-35.44		(Twenty-Five and 44/100 Dollars)							
5	-4		(Four and 00/100 Dollars)							
6	1.87941		One and 87941/100000 Dollars							
7	1.95		One and 95/100 Dollars							
8	1		One and 00/100 Dollars							
9	6.56		Six and 56/100 Dollars							
10	12.12		Twelve and 12/100 Dollars							
11	1000000		One Million and 00/100 Dollars							
12	1000000000		Ten Billion and 00/100 Dollars							
13	1211111111		One Billion One Hundred Eleven Thousand One Hundred Eleven and 00/100 Dollars							

FIGURE 11-17: Examples of the SPELLDOLLARS function.

### A multifunctional function

This example describes a technique that may be helpful in some situations: making a single worksheet function act like multiple functions. For example, the following VBA listing is for a custom function called StatFunction. It takes two arguments: the range (rng) and the operation (op). Depending on the value of op, the function returns a value computed using any of the following worksheet functions: AVERAGE, COUNT, MAX, MEDIAN, MIN, MODE, STDEV, SUM, or VAR.

For example, you can use this function in your worksheet as follows:

`=StatFunction(B1:B24,A24)`

The result of the formula depends on the contents of cell A24, which should be a string such as Average, Count, Max, and so on. You can adapt this technique for other types of functions.

*Function StatFunction(rng, op)*

*Select Case UCase(op)*

*Case "SUM"*

*StatFunction = WorksheetFunction.Sum(rng)*

*Case "AVERAGE"*

*StatFunction = WorksheetFunction.Average(rng)*

*Case "MEDIAN"*

*StatFunction = WorksheetFunction.Median(rng)*

*Case "MODE"*

*StatFunction = WorksheetFunction.Mode(rng)*

*Case "COUNT"*

*StatFunction = WorksheetFunction.Count(rng)*

*Case "MAX"*

*StatFunction = WorksheetFunction.Max(rng)*

*Case "MIN"*

*StatFunction = WorksheetFunction.Min(rng)*

*Case "VAR"*

*StatFunction = WorksheetFunction.Var(rng)*

```
Case "STDEV"  
StatFunction = WorksheetFunction.StDev(rng)  
Case Else  
StatFunction = CVErr(xlErrNA)  
End Select  
End Function
```

### The SheetOffset function

You probably know that Excel's support for 3-D workbooks is limited. For example, if you need to refer to a different worksheet in a workbook, you must include the worksheet's name in your formula. Adding the worksheet name isn't a big problem . . . until you attempt to copy the formula across other worksheets. The copied formulas continue to refer to the original worksheet name, and the sheet references aren't adjusted as they would be in a true 3-D workbook.

The example discussed in this section is a VBA function (named SheetOffset) that enables you to address worksheets in a relative manner. For example, you can refer to cell A1 on the previous worksheet by using this formula:

*=SheetOffset(-1,A1)*

The first argument represents the relative sheet, and it can be positive, negative, or zero. The second argument must be a reference to a single cell. You can copy this formula to other sheets, and the relative referencing will be in effect in all the copied formulas.

The VBA code for the SheetOffset function follows:

```
Function SheetOffset(Offset As Long, Optional Cell As Variant)  
    ' Returns cell contents at Ref, in sheet offset  
    Dim WksIndex As Long, WksNum As Long  
    Dim wks As Worksheet  
    Application.Volatile  
    If IsMissing(Cell) Then Set Cell = Application.Caller  
    WksNum = 1  
    For Each wks In Application.Caller.Parent.Parent.Worksheets  
        If Application.Caller.Parent.Name = wks.Name Then  
            SheetOffset = Worksheets(WksNum + Offset).Range(Cell(1).Address)  
            Exit Function  
        Else  
            WksNum = WksNum + 1  
        End If  
    Next wks  
End Function
```

### Returning the maximum value across all worksheets

If you need to determine the maximum value in cell B1 across a number of worksheets, you would use a formula such as this:

*=MAX(Sheet1:Sheet4!B1)*

This formula returns the maximum value in cell B1 for Sheet1, Sheet4, and all the sheets in between.

But what if you add a new sheet (Sheet5) after Sheet4? Your formula won't adjust automatically, so you need to edit the formula to include the new sheet reference:

`=MAX(Sheet1:Sheet5!B1)`

The MaxAllSheets function, which follows, accepts a single-cell argument and returns the maximum value in that cell across all worksheets in the workbook. The formula that follows, for example, returns the maximum value in cell B1 for all sheets in the workbook:

`=MaxAllSheets(B1)`

If you add a new sheet, you don't need to edit the formula:

```
Function MaxAllSheets(cell)
    Dim MaxVal As Double
    Dim Addr As String
    Dim Wksht As Object
    Application.Volatile
    Addr = cell.Range("A1").Address
    MaxVal = -9.9E+307
    For Each Wksht In cell.Parent.Parent.Worksheets
        If Wksht.Name = cell.Parent.Name And _
            Addr = Application.Caller.Address Then
            ' avoid circular reference
        Else
            If IsNumeric(Wksht.Range(Addr)) Then
                If Wksht.Range(Addr) > MaxVal Then _
                    MaxVal = Wksht.Range(Addr).Value
            End If
        End If
    Next Wksht
    If MaxVal = -9.9E+307 Then MaxVal = 0
    MaxAllSheets = MaxVal
End Function
```

The For Each statement uses the following expression to access the workbook:

`cell.Parent.Parent.Worksheets`

The parent of the cell is a worksheet, and the parent of the worksheet is the workbook. Therefore, the For Each-Next loop cycles among all worksheets in the workbook. The first If statement inside the loop performs a check to see whether the cell being checked is the cell that contains the function. If so, that cell is ignored to avoid a circular reference error.

**Notes:** You can easily modify this function to perform other cross-worksheet calculations, such as minimum, average, sum, and so on.

**Returning an array of nonduplicated random integers**

The function in this section, RandomIntegers, returns an array of nonduplicated integers. The function is intended to be used in a multicell array formula.

`{=RandomIntegers()}`

Select a range and then enter the formula by pressing Ctrl+Shift+Enter. The formula returns an array of nonduplicated integers, arranged randomly. For example, if you enter the formula into a 50-cell range, the formulas will return nonduplicated integers from 1 to 50.

The code for RandomIntegers follows:

```
Function RandomIntegers()  
    Dim FuncRange As Range  
    Dim V() As Variant, ValArray() As Variant  
    Dim CellCount As Double  
    Dim i As Integer, j As Integer  
    Dim r As Integer, c As Integer  
    Dim Temp1 As Variant, Temp2 As Variant  
    Dim RCount As Integer, CCount As Integer  
    ' Create Range object  
    Set FuncRange = Application.Caller  
    ' Return an error if FuncRange is too large  
    CellCount = FuncRange.Count  
    If CellCount > 1000 Then  
        RandomIntegers = CVErr(xlErrNA)  
        Exit Function  
    End If  
    ' Assign variables  
    RCount = FuncRange.Rows.Count  
    CCount = FuncRange.Columns.Count  
    ReDim V(1 To RCount, 1 To CCount)  
    ReDim ValArray(1 To 2, 1 To CellCount)  
    ' Fill array with random numbers  
    ' and consecutive integers  
    For i = 1 To CellCount  
        ValArray(1, i) = Rnd  
        ValArray(2, i) = i  
    Next i  
    ' Sort ValArray by the random number dimension  
    For i = 1 To CellCount  
        For j = i + 1 To CellCount  
            If ValArray(1, i) > ValArray(1, j) Then  
                Temp1 = ValArray(1, j)  
                Temp2 = ValArray(2, j)
```

```

ValArray(1, j) = ValArray(1, i)
ValArray(2, j) = ValArray(2, i)
ValArray(1, i) = Temp1
ValArray(2, i) = Temp2
End If
Next j
Next i
' Put the randomized values into the V array
i = 0
For r = 1 To RCount
For c = 1 To CCount
i = i + 1
V(r, c) = ValArray(2, i)
Next c
Next r
RandomIntegers = V
End Function

```

### Randomizing a range

The RangeRandomize function, which follows, accepts a range argument and returns an array that consists of the input range — in random order:

```

Function RangeRandomize(rng)
Dim V() As Variant, ValArray() As Variant
Dim CellCount As Double
Dim i As Integer, j As Integer
Dim r As Integer, c As Integer
Dim Temp1 As Variant, Temp2 As Variant
Dim RCount As Integer, CCount As Integer
' Return an error if rng is too large
CellCount = rng.Count
If CellCount > 1000 Then
RangeRandomize = CVErr(xlErrNA)
Exit Function
End If
' Assign variables
RCount = rng.Rows.Count
CCount = rng.Columns.Count
ReDim V(1 To RCount, 1 To CCount)
ReDim ValArray(1 To 2, 1 To CellCount)
' Fill ValArray with random numbers
' and values from rng

```

```
For i = 1 To CellCount
    ValArray(1, i) = Rnd
    ValArray(2, i) = rng(i)
Next i
' Sort ValArray by the random number dimension
For i = 1 To CellCount
    For j = i + 1 To CellCount
        If ValArray(1, i) > ValArray(1, j) Then
            Temp1 = ValArray(1, j)
            Temp2 = ValArray(2, j)
            ValArray(1, j) = ValArray(1, i)
            ValArray(2, j) = ValArray(2, i)
            ValArray(1, i) = Temp1
            ValArray(2, i) = Temp2
        End If
    Next j
Next i
' Put the randomized values into the V array
i = 0
For r = 1 To RCount
    For c = 1 To CCount
        i = i + 1
        V(r, c) = ValArray(2, i)
    Next c
Next r
RangeRandomize = V
End Function
```

The code is very similar to that for the RandomIntegers function.

Figure 11-18 shows the function in use. The array formula in B2:B11 is:

*{= RangeRandomize(A2:A11)}*

This formula returns the contents of A2:A11, but in random order.

	A	B	C	D
1	Original	Randomized		
2	Aardvark	Kangaroo		
3	Baboon	Fox		
4	Cat	Hippo		
5	Dog	Cat		
6	Elephant	Iguana		
7	Fox	Lemur		
8	Giraffe	Javelina		
9	Hippo	Baboon		
10	Iguana	Aardvark		
11	Javelina	Elephant		
12	Kangaroo	Giraffe		
13	Lemur	Dog		
14				
15				

FIGURE 11-18: The RangeRandomize function returns the contents of a range, in random order.

## 5.6. Windows API Calls

VBA has the capability to use functions that are stored in Dynamic Link Libraries (DLLs). The examples in this section use common Windows API calls to DLLs.

### Determining file associations

In Windows, many file types are associated with a particular application. This association makes it possible to double-click the file to load it into its associated application.

The following function, named GetExecutable, uses a Windows API call to get the full path to the application associated with a particular file. For example, your system has many files with a .txt extension — one named Readme.txt is probably in your Windows directory right now. You can use the GetExecutable function to determine the full path of the application that opens when the file is double-clicked.

**Notes: Windows API declarations must appear at the top of your VBA module.**

```
Private Declare PtrSafe Function FindExecutableA Lib "shell32.dll" _
    (ByVal lpFile As String, ByVal lpDirectory As String, _
    ByVal lpResult As String) As Long
Function GetExecutable(strFile As String) As String
    Dim strPath As String
    Dim intLen As Integer
    strPath = Space(255)
    intLen = FindExecutableA(strFile, "\", strPath)
    GetExecutable = Trim(strPath)
End Function
```

Figure 11-19 shows the result of calling the GetExecutable function, with an argument of the filename for an MP3 audio file. The function returns the full path of the application that's associated with the file.



FIGURE 11-19: Determining the path and name of the application associated with a particular file.

### Determining disk drive information

VBA doesn't have a way to directly get information about disk drives. But with the assistance of three API functions, you can get just about all the information you need.

Figure 11-20 shows the output from a VBA procedure that identifies all connected drives, determines the drive type, and calculates total space, used space, and free space.

The code is rather lengthy, so I don't list it here, but the interested reader should be able to figure it out by examining the code on the CD-ROM.

	B	C	D	E	F	G
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						

Drive	Type	Total Bytes	Used Bytes	Free Bytes
C:\	Fixed	420,414,020,480	140,444,032,040	280,000,000,000
D:\	Fixed	10,737,414,144	3,737,821,144	6,999,592,960
E:\	CD-ROM	47,418,240	47,418,240	
F:\	CD-ROM			
G:\	Fixed	300,089,586,688	271,551,561,728	28,538,024,960
H:\	Fixed	300,089,586,688	289,034,698,752	11,054,887,936
I:\	Removable			
J:\	Removable			
K:\	Removable	3,956,801,536	884,945,920	3,071,855,616
L:\	Removable			
M:\	Removable			
N:\	Removable			
O:\	Removable			
P:\	Removable			
Q:\	Removable			
R:\	Removable			
S:\	Removable			
T:\	Removable			
U:\	Removable			
V:\	Removable			
W:\	Removable			
X:\	Removable			
Y:\	Removable			
Z:\	Removable			

FIGURE 11-20: Using Windows API functions to get disk drive information.

### Determining default printer information

The example in this section uses a Windows API function to return information about the active printer. The information is contained in a single text string. The example parses the string and displays the information in a more readable format.

```
Private Declare PtrSafe Function GetProfileStringA Lib "kernel32" _
    (ByVal lpAppName As String, ByVal lpKeyName As String, _
    ByVal lpDefault As String, ByVal lpReturnedString As _
    String, ByVal nSize As Long) As Long
Sub DefaultPrinterInfo()
```



```

Dim strLPT As String * 255
Dim Result As String
Call GetProfileStringA _
("Windows", "Device", "", strLPT, 254)
Result = Application.Trim(strLPT)
ResultLength = Len(Result)
Comma1 = InStr(1, Result, ",", 1)
Comma2 = InStr(Comma1 + 1, Result, ",", 1)
' Gets printer's name
Printer = Left(Result, Comma1 - 1)
' Gets driver
Driver = Mid(Result, Comma1 + 1, Comma2 - Comma1 - 1)
' Gets last part of device line
Port = Right(Result, ResultLength - Comma2)
' Build message
Msg = "Printer:" & Chr(9) & Printer & Chr(13)
Msg = Msg & "Driver:" & Chr(9) & Driver & Chr(13)
Msg = Msg & "Port:" & Chr(9) & Port
' Display message
MsgBox Msg, vbInformation, "Default Printer Information"
End Sub

```

Figure 11-21 shows a sample message box returned by this procedure.



FIGURE 11-21: Getting information about the active printer by using a Windows API call.

### Determining video display information

The example in this section uses Windows API calls to determine a system's current video mode for the primary display monitor. If your application needs to display a certain amount of information on one screen, knowing the display size helps you scale the text accordingly. In addition, the code determines the number of monitors. If more than one monitor is installed, the procedure reports the virtual screen size.

```

Declare PtrSafe Function GetSystemMetrics Lib "user32" _
(ByVal nIndex As Long) As Long
Public Const SM_CMONITORS = 80
Public Const SM_CXSCREEN = 0

```

```

Public Const SM_CYSCREEN = 1
Public Const SM_CXVIRTUALSCREEN = 78
Public Const SM_CYVIRTUALSCREEN = 79
Sub DisplayVideoInfo()
Dim numMonitors As Long
Dim vidWidth As Long, vidHeight As Long
Dim virtWidth As Long, virtHeight As Long
Dim Msg As String
numMonitors = GetSystemMetrics(SM_CMONITORS)
vidWidth = GetSystemMetrics(SM_CXSCREEN)
vidHeight = GetSystemMetrics(SM_CYSCREEN)
virtWidth = GetSystemMetrics(SM_CXVIRTUALSCREEN)
virtHeight = GetSystemMetrics(SM_CYVIRTUALSCREEN)
If numMonitors > 1 Then
Msg = numMonitors & " display monitors" & vbCrLf
Msg = Msg & "Virtual screen: " & virtWidth & " X "
Msg = Msg & virtHeight & vbCrLf & vbCrLf
Msg = Msg & "The video mode on the primary display is: "
Msg = Msg & vidWidth & " X " & vidHeight
Else
Msg = Msg & "The video display mode: "
Msg = Msg & vidWidth & " X " & vidHeight
End If
MsgBox Msg
End Sub

```

Figure 11-22 shows the message box returned by this procedure when running on a dual-monitor system.



FIGURE 11-22: Using a Windows API call to determine the video display mode.

### Reading from and writing to the Registry

Most Windows applications use the Windows Registry database to store settings. (See Chapter 4 for some additional information about the Registry.) Your VBA procedures can read values from the Registry and write new values to the Registry. Doing so requires the following Windows API declarations:

```

Private Declare PtrSafe Function RegOpenKeyA Lib "ADVAPI32.DLL" _
(ByVal hKey As Long, ByVal sSubKey As String, _

```

*ByRef hkeyResult As Long) As Long*

*Private Declare PtrSafe Function RegCloseKey Lib "ADVAPI32.DLL" \_*

*(ByVal hKey As Long) As Long*

*Private Declare PtrSafe Function RegSetValueExA Lib "ADVAPI32.DLL"*

*(ByVal hKey As Long, ByVal sValueName As String, \_*

*ByVal dwReserved As Long, ByVal dwType As Long, \_*

*ByVal sValue As String, ByVal dwSize As Long) As Long*

*Private Declare PtrSafe Function RegCreateKeyA Lib "ADVAPI32.DLL"*

*(ByVal hKey As Long, ByVal sSubKey As String, \_*

*ByRef hkeyResult As Long) As Long*

*Private Declare PtrSafe Function RegQueryValueExA Lib "ADVAPI32.DLL" \_*

*(ByVal hKey As Long, ByVal sValueName As String, \_*

*ByVal dwReserved As Long, ByRef lValueType As Long, \_*

*ByVal sValue As String, ByRef lResultLen As Long) As Long*

### Reading from the Registry

The GetRegistry function returns a setting from the specified location in the Registry. It takes three arguments:

- RootKey: A string that represents the branch of the Registry to address. This string can be one of the following:
  - HKEY\_CLASSES\_ROOT
  - HKEY\_CURRENT\_USER
  - HKEY\_LOCAL\_MACHINE
  - HKEY\_USERS
- HKEY\_CURRENT\_CONFIG
- Path: The full path of the Registry category being addressed.
- RegEntry: The name of the setting to retrieve.

Here's an example. If you'd like to find which graphic file, if any, is being used for the desktop wallpaper, you can call GetRegistry as follows. (Notice that the arguments aren't case-sensitive.)

*RootKey = "hkey\_current\_user"*

*Path = "Control Panel\Desktop"*

*RegEntry = "Wallpaper"*

*MsgBox GetRegistry(RootKey, Path, RegEntry), \_*

*vbInformation, Path & "RegEntry"*

The message box will display the path and filename of the graphic file (or an empty string if wallpaper isn't used).

**Writing to the Registry**

The WriteRegistry function writes a value to the Registry at a specified location. If the operation is successful, the function returns True; otherwise, it returns False. WriteRegistry takes the following arguments (all of which are strings):

- RootKey: A string that represents the branch of the Registry to address. This string may be one of the following:
  - HKEY\_CLASSES\_ROOT
  - HKEY\_CURRENT\_USER
  - HKEY\_LOCAL\_MACHINE
  - HKEY\_USERS
  - HKEY\_CURRENT\_CONFIG
- Path: The full path in the Registry. If the path doesn't exist, it is created.
- RegEntry: The name of the Registry category to which the value will be written. If it doesn't exist, it is added.
- RegVal: The value that you're writing.

Here's an example that writes a value representing the time and date Excel was started to the Registry. The information is written in the area that stores Excel's settings.

```
Sub Workbook_Open()  
RootKey = "hkey_current_user"  
Path = "software\microsoft\office\14.0\excel\LastStarted"  
RegEntry = "DateTime"  
RegVal = Now()  
If WriteRegistry(RootKey, Path, RegEntry, RegVal) Then  
msg = RegVal & " has been stored in the registry."  
Else msg = "An error occurred"  
End If  
MsgBox msg  
End Sub
```

If you store this routine in the ThisWorkbook module in your personal macro workbook, the setting is automatically updated whenever you start Excel.

## 6. Custom Dialog Box Alternatives

### 6.1. Before You Create That UserForm

Dialog boxes are, perhaps, the most important user interface element in Windows programs. Virtually every Windows program uses them, and most users have a good understanding of how they work. Excel developers implement custom dialog boxes by creating UserForms. However, VBA provides the means to display some built-in dialog boxes, with minimal programming required.

Before I get into the nitty-gritty of creating UserForms (beginning with Chapter 13), you might find it helpful to understand some of Excel's built-in tools that display dialog boxes. The sections that follow describe various dialog boxes that you can display without creating a UserForm.

### 6.2. Using an Input Box

An input box is a simple dialog box that allows the user to make a single entry. For example, you can use an input box to let the user enter text or a number or even select a range. You can generate an `InputBox` in two ways: by using a VBA function and by using a method of the `Application` object.

#### The VBA `InputBox` function

The syntax for VBA's `InputBox` function is

`InputBox(prompt[,title][,default][,xpos][,ypos][,helpfile, context])`

- **prompt:** Required. The text displayed in the `InputBox`.
- **title:** Optional. The caption of the `InputBox` window.
- **default:** Optional. The default value to be displayed in the dialog box.
- **xpos, ypos:** Optional. The screen coordinates of the upper-left corner of the window.
- **helpfile, context:** Optional. The help file and help topic.

The `InputBox` function prompts the user for a single piece of information. The function always returns a string, so your code may need to convert the results to a value.

The prompt may consist of up to 1,024 characters. In addition, you can provide a title for the dialog box and a default value and specify its position on the screen. You can also specify a custom Help topic; if you do, the input box includes a Help button.

The following example, which generates the dialog box shown in Figure 12-1, uses the VBA `InputBox` function to ask the user for his full name. The code then extracts the first name and displays a greeting in a message box.



FIGURE 12-1: VBA's `InputBox` function at work.

```
Sub GetName()  
    Dim UserName As String  
    Dim FirstSpace As Integer  
    Do Until UserName <> ""
```

```

UserName = InputBox("Enter your full name: ", _
    "Identify Yourself")
Loop
FirstSpace = InStr(UserName, " ")
If FirstSpace <> 0 Then
    UserName = Left(UserName, FirstSpace - 1)
End If
MsgBox "Hello " & UserName
End Sub

```

Notice that this InputBox function is written in a Do Until loop to ensure that something is entered when the input box appears. If the user clicks Cancel or doesn't enter any text, UserName contains an empty string, and the input box reappears. The procedure then attempts to extract the first name by searching for the first space character (by using the InStr function) and then using the Left function to extract all characters before the first space. If a space character isn't found, the entire name is used as entered.

As I mentioned, the InputBox function always returns a string. If the string returned by the InputBox function looks like a number, you can convert it to a value by using VBA's Val function. Or you can use Excel's InputBox method, which I describe in the next section.

Figure 12-2 shows another example of the VBA InputBox function. The user is asked to fill in the missing word. This example also illustrates the use of named arguments. The prompt text is retrieved from a worksheet cell and is assigned to a variable (p).



FIGURE 12-2: Using VBA's InputBox function with a long prompt.

```

Sub GetWord()
    Dim TheWord As String
    Dim p As String
    Dim t As String
    p = Range("A1")
    t = "What's the missing word?"
    TheWord = InputBox(prompt:=p, Title:=t)
    If UCase(TheWord) = "BATTLEFIELD" Then
        MsgBox "Correct."
    End If
End Sub

```

```

Else
MsgBox "That is incorrect."
End If
End Sub

```

### The Excel InputBox method

Using Excel's InputBox method offers three advantages over VBA's InputBox function:

- You can specify the data type returned.
- The user can specify a worksheet range by dragging in the worksheet.
- Input validation is performed automatically.

The syntax for the Excel InputBox method is

InputBox(Prompt [,Title][,Default][,Left][,Top][,HelpFile, HelpContextID][,Type])

- **Prompt: Required.** The text displayed in the input box.
- **Title:** Optional. The caption in the input box window.
- **Default:** Optional. The default value to be returned by the function if the user enters nothing.
- **Left, Top:** Optional. The screen coordinates of the upper-left corner of the window.
- **HelpFile, HelpContextID:** Optional. The Help file and Help topic.
- **Type:** Optional. A code for the data type returned, as listed in Table 12-1.

**Table 12-1: Codes to Determine the Data Type Returned by Excel's Inputbox Method**

Code	Meaning
0	A formula
1	A number
2	A string (text)
4	A logical value (True or False)
8	A cell reference, as a range object
16	An error value, such as #N/A
64	An array of values

Excel's InputBox method is quite versatile. To allow more than one data type to be returned, use the sum of the pertinent codes. For example, to display an input box that can accept text or numbers, set type equal to 3 (that is, 1 + 2, or number plus text). If you use 8 for the type argument, the user can enter a cell or range address (or a named cell or range) manually or point to a range in the worksheet.

The EraseRange procedure, which follows, uses the InputBox method to allow the user to select a range to erase (see Figure 12-3). The user can either type the range address manually or use the mouse to select the range in the sheet.

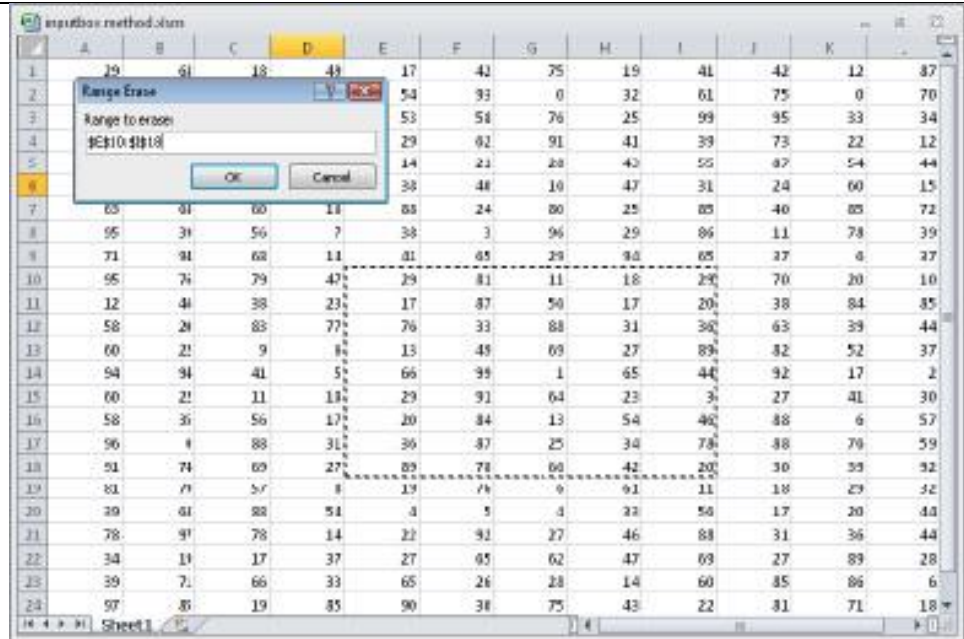


FIGURE 12-3: Using the InputBox method to specify a range.

The InputBox method with a type argument of 8 returns a Range object (note the Set keyword). This range is then erased (by using the Clear method). The default value displayed in the input box is the current selection's address. The On Error statement ends the procedure if the input box is canceled.

```
Sub EraseRange()
    Dim UserRange As Range
    On Error GoTo Canceled
    Set UserRange = Application.InputBox _
        (Prompt:="Range to erase:", _
        Title:="Range Erase", _
        Default:=Selection.Address, _
        Type:=8)
    UserRange.Clear
    UserRange.Select
    Canceled:
End Sub
```

Yet another advantage of using Excel's InputBox method is that Excel performs input validation automatically. In the GetRange example, if you enter something other than a range address, Excel displays an informative message and lets the user try again (see Figure 12-4).



FIGURE 12-4: Excel's InputBox method performs validation automatically.



### 6.3. The VBA MsgBox Function

VBA's MsgBox function is an easy way to display a message to the user or to get a simple response (such as OK or Cancel). I use the MsgBox function in many of the examples in this book as a way to display a variable's value.

The official syntax for MsgBox is as follows:

MsgBox(prompt[,buttons][,title][,helpfile, context])

- **prompt:** Required. The text displayed in the message box.
- **buttons:** Optional. A numeric expression that determines which buttons and icon are displayed in the message box. See Table 12-2.
- **title:** Optional. The caption in the message box window.
- **helpfile, context:** Optional. The helpfile and Help topic.

You can easily customize your message boxes because of the flexibility of the buttons argument. (Table 12-2 lists the many constants that you can use for this argument.) You can specify which buttons to display, whether an icon appears, and which button is the default.

**Table 12-2: Constants Used for Buttons in the MsgBox Function**

Constant	Value	Description
vbOKOnly	0	Display OK button only.
vbOKCancel	1	Display OK and Cancel buttons.
vbAbortRetryIgnore	2	Display Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Display Yes, No, and Cancel buttons.
vbYesNo	4	Display Yes and No buttons.
vbRetryCancel	5	Display Retry and Cancel buttons.
vbCritical	16	Display Critical Message icon.
vbQuestion	32	Display Warning Query icon.
vbExclamation	48	Display Warning Message icon.
vbInformation	64	Display Information Message icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbSystemModal	4096	All applications are suspended until the user responds to the message box (might not work under all conditions).
vbMsgBoxHelpButton	16384	Display a Help button. However, there is no way to display any help if the button is clicked.

You can use the MsgBox function by itself (to simply display a message) or assign its result to a variable. When you use the MsgBox function to return a result, the value represents the button clicked by the user. The following example displays a message and an OK button, but doesn't return a result:

```
Sub MsgBoxDemo()
    MsgBox "Macro finished with no errors."
End Sub
```

To get a response from a message box, you can assign the results of the MsgBox function to a variable. In the following code, I use some built-in constants (described in Table 12-3) to make it easier to work with the values returned by MsgBox:

```
Sub GetAnswer()
    Dim Ans As Integer
    Ans = MsgBox("Continue?", vbYesNo)
    Select Case Ans
        Case vbYes
            ' ...[code if Ans is Yes]...
        Case vbNo
            ' ...[code if Ans is No]...
    End Select
End Sub
```

**Table 12-3: Constants Used for MsgBox Return Value**

Constant	Value	Button Clicked
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

The variable returned by the MsgBox function is an Integer data type. Actually, you don't even need to use a variable to utilize the result of a message box. The following procedure is another way of coding the GetAnswer procedure:

```
Sub GetAnswer2()
    If MsgBox("Continue?", vbYesNo) = vbYes Then
        ' ...[code if Ans is Yes]...
    Else
        ' ...[code if Ans is No]...
    End If
```

*End Sub*

The following function example uses a combination of constants to display a message box with a Yes button, a No button, and a question mark icon; the second button is designated as the default button (see Figure 12-5). For simplicity, I assigned these constants to the Config variable.



FIGURE 12-5: The buttons argument of the MsgBox function determines which buttons appear.

*Private Function ContinueProcedure() As Boolean*

*Dim Config As Integer*

*Dim Ans As Integer*

*Config = vbYesNo + vbQuestion + vbDefaultButton2*

*Ans = MsgBox("An error occurred. Continue?", Config)*

*If Ans = vbYes Then ContinueProcedure = True \_*

*Else ContinueProcedure = False*

*End Function*

You can call the ContinueProcedure function from another procedure. For example, the following statement calls the ContinueProcedure function (which displays the message box). If the function returns False (that is, the user selects No), the procedure ends. Otherwise, the next statement would be executed.

*If Not ContinueProcedure() Then Exit Sub*

The width of the message box depends on your video resolution. If you'd like to force a line break in the message, use the vbCrLf (or vbNewLine) constant in the text. The following example displays the message in three lines. Figure 12-6 shows how it looks.



FIGURE 12-6: Splitting a message into multiple lines.

*Sub MultiLine()*

*Dim Msg As String*

*Msg = "This is the first line." & vbCrLf & vbCrLf*

*Msg = Msg & "This is the second line." & vbCrLf*

*Msg = Msg & "And this is the last line."*

*MsgBox Msg*

*End Sub*

You can also insert a tab character by using the vbTab constant. The following procedure uses a message box to display the values in a 13 x 3 range of cells in A1:C13 (see Figure 12-7). It separates the columns by using a vbTab constant and inserts a new line by using the vbCrLf constant. The MsgBox function accepts a maximum string length of 1,023 characters, which will limit the number of cells that you can display. Also, note that the tab stops are fixed, so if a cell contains more than 11 characters, the columns won't be aligned.

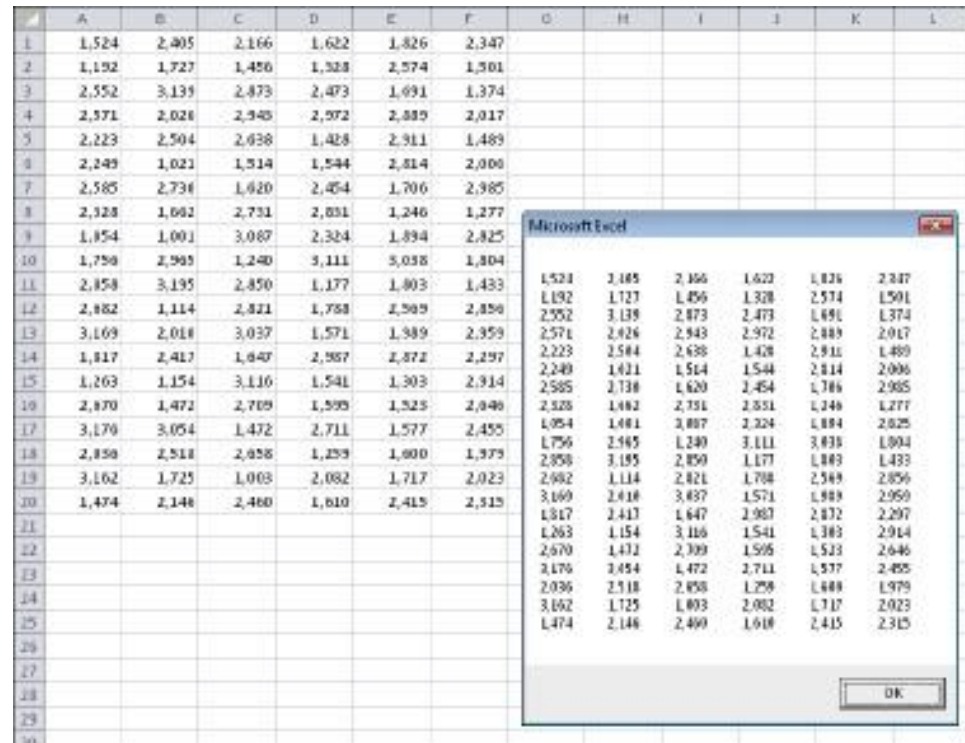


FIGURE 12-7: This message box displays text with tabs and line breaks.

```

Sub ShowRange()
    Dim Msg As String
    Dim r As Integer, c As Integer
    Msg = ""
    For r = 1 To 12
        For c = 1 To 3
            Msg = Msg & Cells(r, c).Text
            If c <> 3 Then Msg = Msg & vbTab
        Next c
        Msg = Msg & vbCrLf
    Next r
    MsgBox Msg
End Sub

```

## 7. Introducing UserForms

### 7.1. How Excel Handles Custom Dialog Boxes

The examples in this section demonstrate how to manipulate worksheet ranges with VBA.

Specifically, I provide examples of copying a range, moving a range, selecting a range, identifying types of information in a range

Excel makes creating custom dialog boxes for your applications relatively easy. In fact, you can duplicate the look and feel of many of Excel's dialog boxes.

Excel developers have always had the ability to create custom dialog boxes for their applications. Beginning with Excel 97, things changed substantially — UserForms replaced the clunky old dialog sheets. UserForms are much easier to work with, and they offer many additional capabilities. Even though UserForms haven't been upgraded over the years, you'll find that this feature works well and is very flexible.

A custom dialog box is created on a UserForm, and you access UserForms in the Visual Basic Editor (VBE).

Following is the typical sequence that you'll follow when you create a UserForm:

1. Insert a new UserForm into your workbook's VB Project.
2. Add controls to the UserForm.
3. Adjust some of the properties of the controls that you added.
4. Write event-handler procedures for the controls.

These procedures, which are located in the code window for the UserForm, are executed when various events (such as a button click) occur.

5. Write a procedure that will display the UserForm.
6. This procedure will be located in a VBA module (not in the code module for the UserForm).
7. Add a way to make it easy for the user to execute the procedure you created in Step 5.

You can add a button to a worksheet, a Ribbon command, and so on.

### 7.2. Inserting a New UserForm

To insert a new UserForm, activate the VBE (press Alt+F11), select your workbook's project from the Project window, and then choose Insert⇒UserForm. UserForms have default names like UserForm1, UserForm2, and so on.

**Notes; You can change the name of a UserForm to make it easier to identify. Select the form and use the Properties window to change the Name property. (Press F4 if the Properties window isn't displayed.) Figure 13-1 shows the Properties window when an empty UserForm is selected.**

A workbook can have any number of UserForms, and each UserForm holds a single custom dialog box.

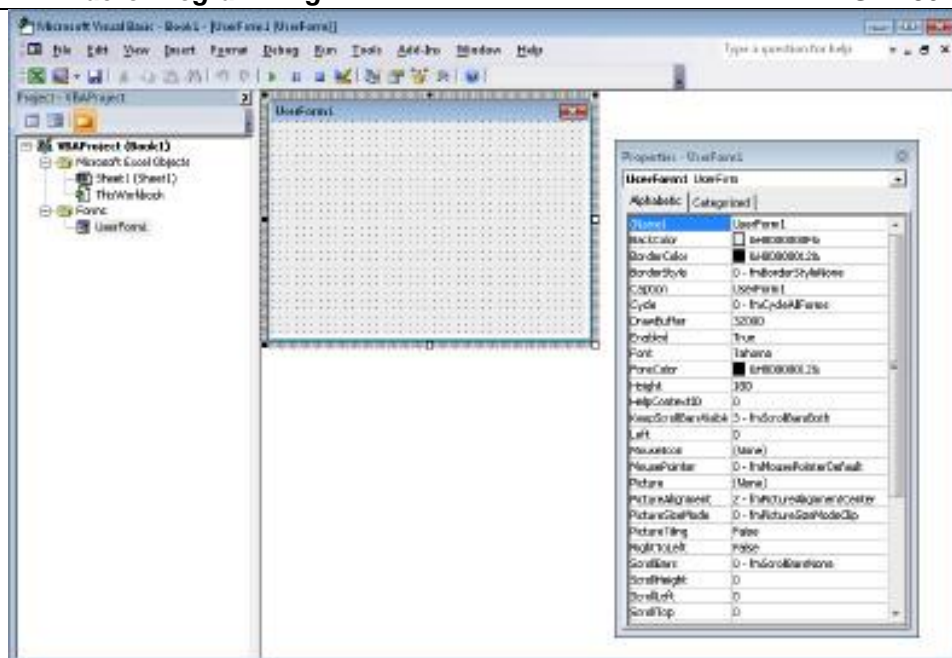


FIGURE 13-1: The Properties window for an empty UserForm.

### 7.3. Adding Controls to a UserForm

To add controls to a UserForm, use the Toolbox. (The VBE doesn't have menu commands that add controls.) If the Toolbox isn't displayed, choose View⇒Toolbox. Figure 13-2 shows the Toolbox. The Toolbox is a floating window, so you can move it to a convenient location.



FIGURE 13-2: Use the Toolbox to add controls to a UserForm.

Click the Toolbox button that corresponds to the control that you want to add and then click inside the dialog box to create the control (using its default size). Or you can click the control and then drag in the dialog box to specify the dimensions for the control.

When you add a new control, it's assigned a name that combines the control type with the numeric sequence for that type of control. For example, if you add a `CommandButton` control to an empty `UserForm`, it's named `CommandButton1`. If you then add a second `CommandButton` control, it's named `CommandButton2`.

**Notes: Renaming all the controls that you'll be manipulating with your VBA code is a good idea. Doing so lets you refer to meaningful names (such as ProductListBox) rather than generic names (such as ListBox1). To change the name of a control, use the Properties window in the VBE. Just select the object and change the Name property.**

## 7.4. Toolbox Controls

In the sections that follow, I briefly describe the controls available to you in the Toolbox.

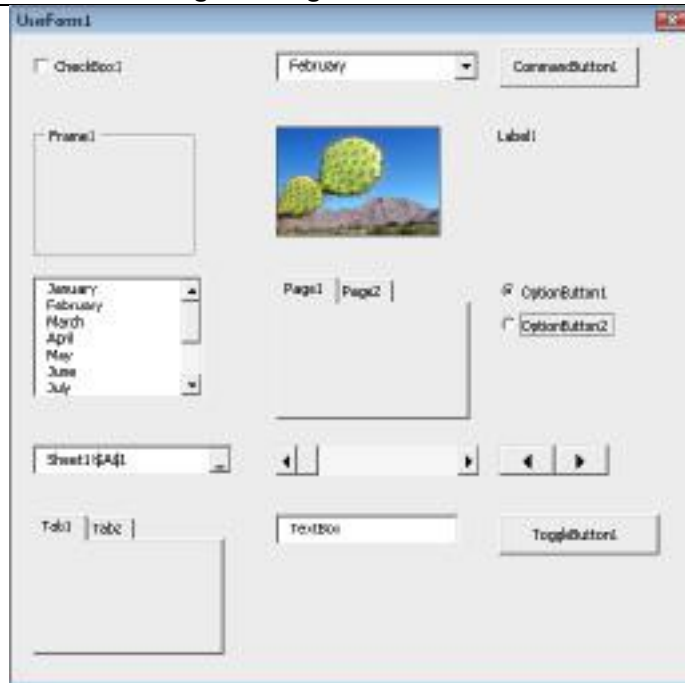


FIGURE 13-3: This UserForm has one of each of the 15 controls.

### CheckBox

A CheckBox control is useful for getting a binary choice: yes or no, true or false, on or off, and so on. When a CheckBox is checked, it has a value of True; when it's not checked, the CheckBox value is False.

### ComboBox

A ComboBox control presents a list of items in a drop-down box and displays only one item at a time. Unlike a ListBox control, you can set up a ComboBox to allow the user to enter a value that doesn't appear in the list of items.

### CommandButton

Every dialog box that you create will probably have at least one CommandButton control. Usually, your UserForms will have one CommandButton labeled OK and another labeled Cancel.

### Frame

A Frame control is used to enclose other controls. You enclose controls either for aesthetic purposes or to logically group a set of controls. A frame is particularly useful when the dialog box contains more than one set of OptionButton controls.

### Image

You can use an Image control to display a graphic image, which can come from a file or can be pasted from the Clipboard. You may want to use an Image control to display your company's logo in a dialog box. The graphics image is stored in the workbook. That way, if you distribute your workbook to someone else, you don't have to include a copy of the graphics file.

**Label**

A Label control simply displays text in your dialog box.

**ListBox**

The ListBox control presents a list of items, and the user can select an item (or multiple items). ListBox controls are very flexible. For example, you can specify a worksheet range that holds the ListBox items, and this range can consist of multiple columns. Or you can fill the ListBox with items by using VBA.

**MultiPage**

A MultiPage control lets you create tabbed dialog boxes, like the Format Cells dialog box. By default, a MultiPage control has two pages, but you can add any number of additional pages.

**OptionButton**

OptionButton controls are useful when the user needs to select one item from a small number of choices. OptionButtons are always used in groups of at least two. When one OptionButton is selected, the other OptionButtons in its group are deselected.

If your UserForm contains more than one set of OptionButtons, the OptionButtons in each set must share a unique GroupName property value. Otherwise, all OptionButtons become part of the same set. Alternatively, you can enclose the OptionButtons in a Frame control, which automatically groups the OptionButtons contained in the frame.

**RefEdit**

The RefEdit control is used when you need to let the user select a range in a worksheet. This control accepts a typed range address or a range address generated by pointing to the range in a worksheet.

**ScrollBar**

The ScrollBar control is similar to a SpinButton control. The difference is that the user can drag the ScrollBar button to change the control's value in larger increments. The ScrollBar control is most useful for selecting a value that extends across a wide range of possible values.

**SpinButton**

The SpinButton control lets the user select a value by clicking either of two arrows: one to increase the value and the other to decrease the value. A SpinButton is often used in conjunction with a TextBox control or Label control, which displays the current value of the SpinButton. A SpinButton can be oriented horizontally or vertically.

**TabStrip**

A TabStrip control is similar to a MultiPage control, but it's not as easy to use. A TabStrip control, unlike a MultiPage control, doesn't serve as a container for



other objects. Generally, you'll find that the MultiPage control is much more versatile.

### TextBox

A TextBox control lets the user type text or a value.

### Using controls on a worksheet

You can embed many of the UserForm controls directly into a worksheet. You can access these controls by using Excel's Developer⇒Controls⇒Insert command. Adding such controls to a worksheet requires much less effort than creating a UserForm. In addition, you may not have to create any macros because you can link a control to a worksheet cell. For example, if you insert a CheckBox control on a worksheet, you can link it to a particular cell by setting its LinkedCell property. When the CheckBox is checked, the linked cell displays TRUE. When the CheckBox is unchecked, the linked cell displays FALSE.

The accompanying figure shows a worksheet that contains some ActiveX controls. This workbook, named *activex worksheet controls.xlsx*, is available on the companion CD-ROM. The workbook uses linked cells and contains no macros.

Adding controls to a worksheet can be a bit confusing because controls can come from two sources:

- Form controls: These controls are insertable objects.
- ActiveX controls: These controls are a subset of those that are available for use on UserForms.

You can use the controls from either of these sources, but it's important that you understand the distinctions between them. The Form controls work much differently than the ActiveX controls.

When you add an ActiveX control to a worksheet, Excel goes into design mode. In this mode, you can adjust the properties of any controls on your worksheet, add or edit event-handler procedures for the control, or change its size or position. To display the Properties window for an ActiveX control, use the Developer⇒Controls⇒Properties command.

For simple buttons, I often use the Button control from the Form controls because it lets me attach any macro to it. If I use a CommandButton control from the ActiveX controls, clicking it will execute its event-handler procedure (for example, CommandButton1\_Click) in the code module for the Sheet object — you can't attach just any macro to it.

When Excel is in design mode, you can't try out the controls. To test the controls, you must exit design mode by clicking the Developer⇒Controls⇒Design mode button (which is a toggle).

### ToggleButton

A ToggleButton control has two states: on and off. Clicking the button toggles between these two states, and the button changes its appearance. Its value is either True (pressed) or False (not pressed). I never use this control because I think a CheckBox is much clearer.

## 7.5. How Excel Handles Custom Dialog Boxes Adjusting UserForm Controls

After you place a control in a UserForm, you can move and resize the control by using standard mouse techniques.

**Notes:** You can select multiple controls by Shift-clicking or by clicking and dragging to lasso a group of controls.

A UserForm can contain vertical and horizontal gridlines (displayed as dots) that help you align the controls that you add. When you add or move a control, it snaps to the grid to help you line up the controls. If you don't like to see these gridlines, you can turn them off by choosing Tools⇒Options in the VBE. In the Options dialog box, select the General tab and set your desired options in the Form Grid Settings section.

The Format menu in the VBE window provides several commands to help you precisely align and space the controls in a dialog box. Before you use these commands, select the controls that you want to work with. These commands work just as you'd expect, so I don't explain them here. Figure 13-4 shows a dialog box with several OptionButton controls about to be aligned. Figure 13-5 shows the controls after being aligned and assigned equal vertical spacing.

**Notes:** When you select multiple controls, the last control that you select appears with white handles rather than the normal black handles. The control with the white handles is used as the basis for sizing or positioning.

## 7.6. Adjusting a Control's Properties

Every control has a number of properties that determine how the control looks and behaves. You can change a control's properties, as follows:

- At design time when you're developing the UserForm. You use the Properties window to make design time changes.
- During runtime when the UserForm is being displayed for the user. You use VBA instructions to change a control's properties at runtime.

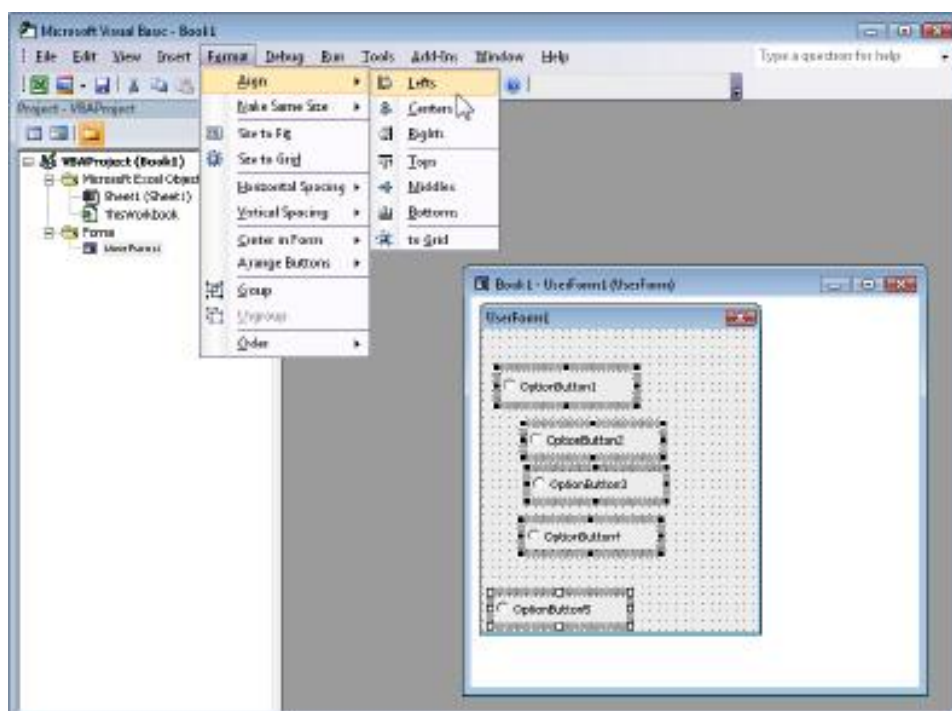


FIGURE 13-4: Use the Format⇒Align command to change the alignment of controls.

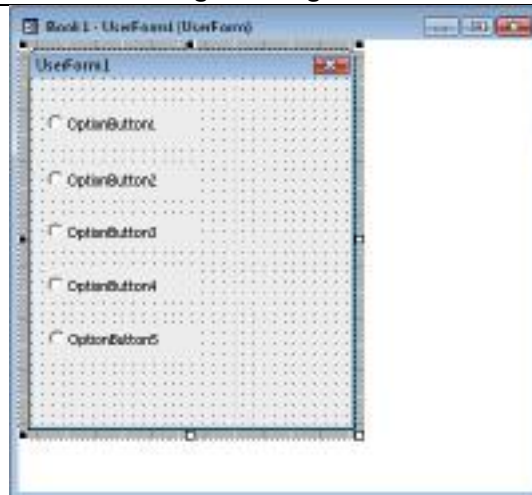


FIGURE 13-5: The OptionButton controls, aligned and evenly spaced.

### Using the Properties window

In the VBE, the Properties window adjusts to display the properties of the selected item (which can be a control or the UserForm itself). In addition, you can select a control from the drop-down list at the top of the Properties window. Figure 13-6 shows the Properties window for an OptionButton control.

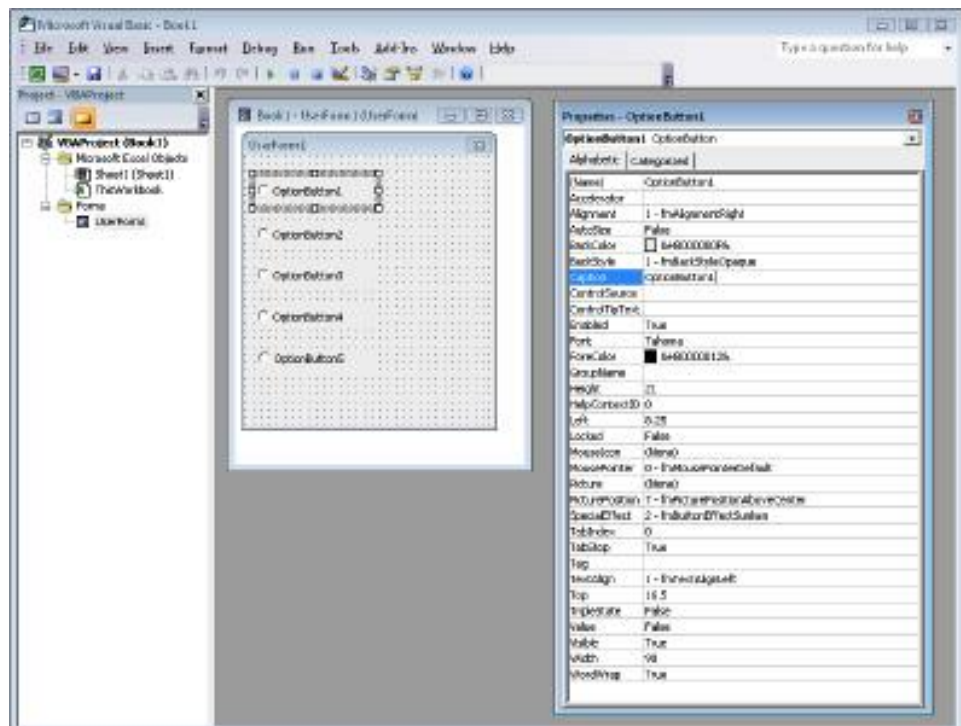


FIGURE 13-6: The Properties window for an OptionButton control.

**Notes:** The Properties window has two tabs. The Alphabetic tab displays the properties for the selected object in alphabetical order. The Categorized tab displays them grouped into logical categories. Both tabs contain the same properties but in a different order.

To change a property, just click it and specify the new property. Some properties can take on a finite number of values, selectable from a list. If so, the Properties window will display a button with a downward-pointing arrow when that property

is selected. Click the button, and you'll be able to select the property's value from the list. For example, the TextAlign property can have any of the following values: 1 - fmTextAlignLeft, 2 - fmTextAlignCenter, or 3 - fmTextAlignRight.

A few properties (for example, Font and Picture) display a small button with an ellipsis when selected. Click the button to display a dialog box associated with the property.

The Image control Picture property is worth mentioning because you can either select a graphic file that contains the image or paste an image from the Clipboard. When pasting an image, first copy it to the Clipboard; then select the Picture property for the Image control, and press Ctrl+V to paste the Clipboard contents.

**Notes:** If you select two or more controls at once, the Properties window displays only the properties that are common to the selected controls.

**Notes:** The UserForm itself has many properties that you can adjust. Some of these properties are then used as defaults for controls that you add to the UserForm. For example, if you change the UserForm Font property, all controls added to the UserForm will use that font. Note, however, that controls already on the UserForm aren't affected.

### Common properties

Although each control has its own unique set of properties, many controls have some common properties. For example, every control has a Name property and properties that determine its size and position (Height, Width, Left, and Right).

If you're going to manipulate a control by using VBA, it's an excellent idea to provide a meaningful name for the control. For example, the first OptionButton that you add to a UserForm has a default name of OptionButton1. You refer to this object in your code with a statement such as the following:

```
OptionButton1.Value = True
```

But if you give the OptionButton a more meaningful name (such as obLandscape), you can use a statement such as this one:

```
obLandscape.Value = True
```

**Notes:** Many people find it helpful to use a name that also identifies the type of object. In the preceding example, I use ob as the prefix to identify the control as an OptionButton. I'm not aware of any standard prefixes, so feel free to invent your own.

You can adjust the properties of several controls at once. For example, you might have several OptionButtons that you want left-aligned. You can simply select all the OptionButtons and then change the Left property in the Properties box. All the selected controls will then take on that new Left property value.

The best way to learn about the various properties for a control is to use the Help system. Simply click on a property in the Property window and press F1. Figure 13-7 shows an example of the type of help provided for a property.

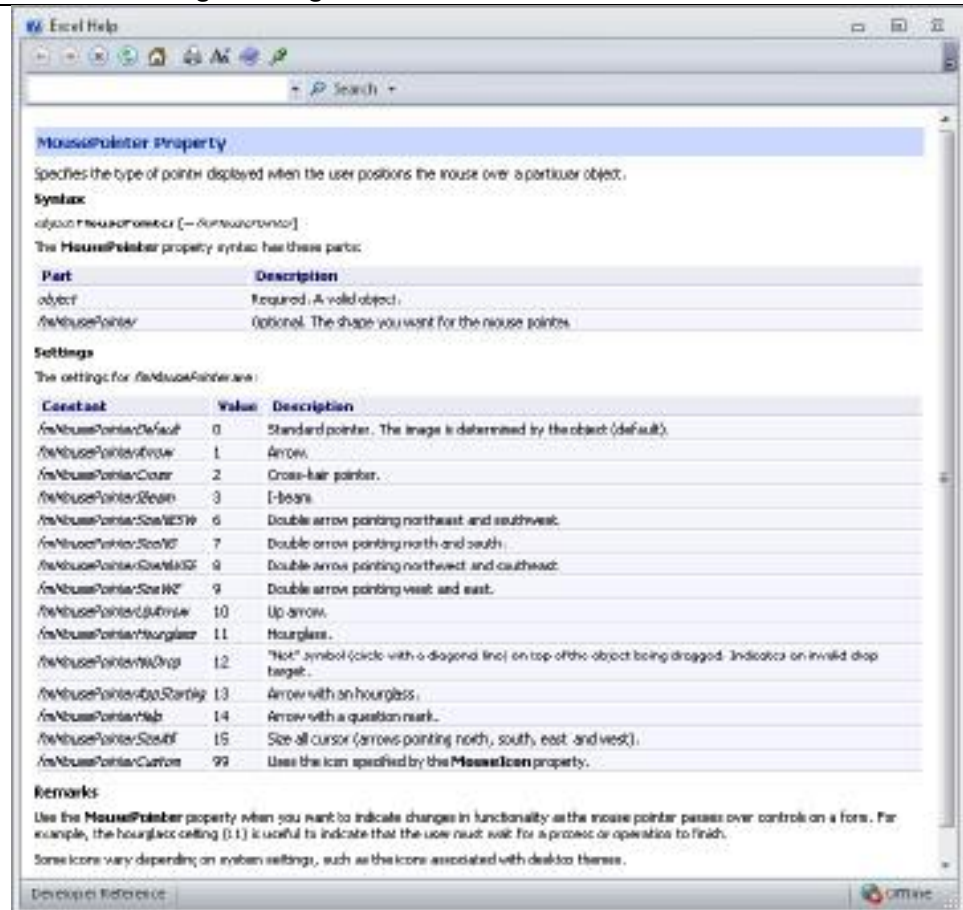


FIGURE 13-7: The Help system provides information about each property for every control.

### Accommodating keyboard users

Many users prefer to navigate through a dialog box by using the keyboard: The Tab and Shift+Tab keystrokes cycle through the controls, and pressing a hot key (an underlined letter) operates the control. To make sure that your dialog box works properly for keyboard users, you must be mindful of two issues: tab order and accelerator keys.

### Changing the tab order of controls

The tab order determines the sequence in which the controls are activated when the user presses Tab or Shift+Tab. It also determines which control has the initial focus. If a user is entering text into a TextBox control, for example, the TextBox has the focus. If the user clicks an OptionButton, the OptionButton has the focus. The control that's first in the tab order has the focus when a dialog box is first displayed.

To set the tab order of your controls, choose View⇒Tab Order. You can also right-click the UserForm and choose Tab Order from the shortcut menu. In either case, Excel displays the Tab Order dialog box, as shown in Figure 13-8. The Tab Order dialog box lists all the controls, the sequence of which corresponds to the order in which controls pass the focus between each other in the UserForm. To move a control, select it and click the arrow keys up or down. You can choose more than one control (click while pressing Shift or Ctrl) and move them all at once.

Alternatively, you can set an individual control's position in the tab order via the Properties window. The first control in the tab order has a `TabIndex` property of

0. Changing the TabIndex property for a control may also affect the TabIndex property of other controls. These adjustments are made automatically to ensure that no control has a TabIndex greater than the number of controls. If you want to remove a control from the tab order, set its TabStop property to False.

**Notes: Some controls, such as Frame and MultiPage, act as containers for other controls. The controls inside a container have their own tab order. To set the tab order for a group of OptionButtons inside a Frame control, select the Frame control before you choose the View⇒Tab Order command.**

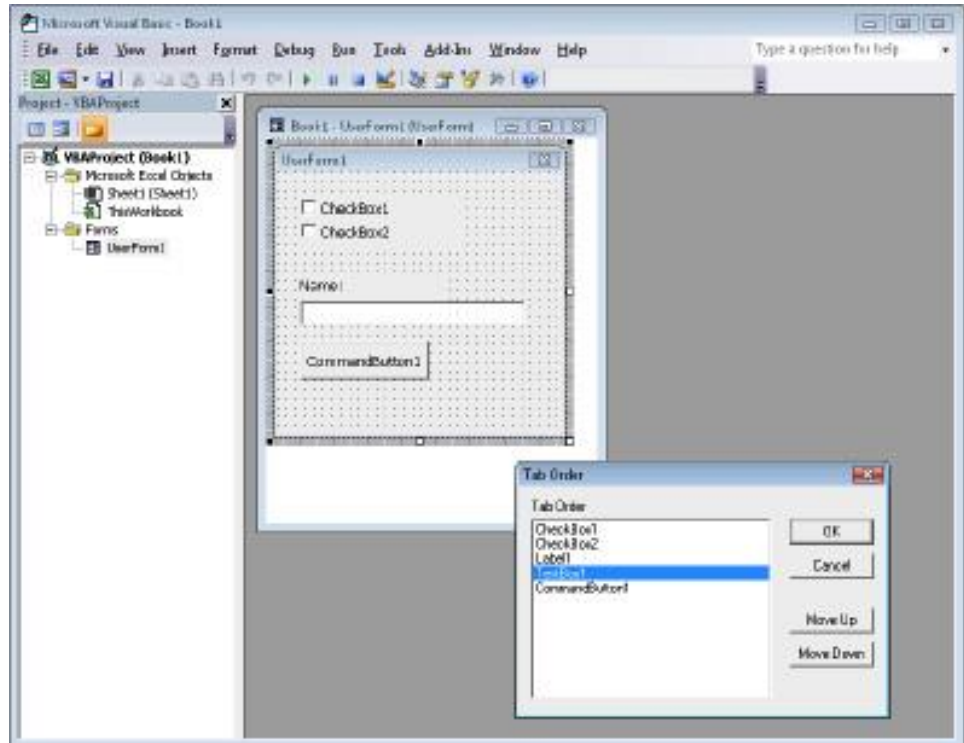


FIGURE 13-8: Use the Tab Order dialog box to specify the tab order of the controls.

### Testing a UserForm

You'll usually want to test your UserForm while you're developing it. You can test a UserForm in three ways without actually calling it from a VBA procedure:

- Choose the Run⇒Run Sub/UserForm command.
- Press F5.
- Click the Run Sub/UserForm button on the Standard toolbar.

These three techniques all trigger the UserForm's Initialize event. When a dialog box is displayed in this test mode, you can try out the tab order and the accelerator keys.

### Setting hot keys

You can assign an accelerator key, or hot key, to most dialog box controls. An accelerator key allows the user to access the control by pressing Alt+ the hot key. Use the Accelerator property in the Properties window for this purpose.

## 7.7. Displaying a UserForm

To display a UserForm from VBA, you create a procedure that uses the Show method of the UserForm object. If your UserForm is named UserForm1, the following procedure displays the dialog box on that form:

```
Sub ShowForm()  
    UserForm1.Show  
End Sub
```

This procedure must be located in a standard VBA module and not in the code module for the UserForm.

When the UserForm is displayed, it remains visible on-screen until it's dismissed. Usually, you'll add a CommandButton control to the UserForm that executes a procedure that dismisses the UserForm. The procedure can either unload the UserForm (with the Unload command) or hide the UserForm (with the Hide method of the UserForm object). This concept will become clearer as you work through various examples in this and subsequent chapters.

### Displaying a modeless UserForm

By default, UserForms are displayed modally. This means that the UserForm must be dismissed before the user can do anything in the worksheet. You can also display a modeless UserForm. When a modeless UserForm is displayed, the user can continue working in Excel, and the UserForm remains visible. To display a modeless UserForm, use the following syntax:

```
UserForm1.Show vbModeless
```

### Displaying a UserForm based on a variable

In some cases, you may have several UserForms, and your code makes a decision regarding which of them to display. If the name of the UserForm is stored as a string variable, you can use the Add method to add the UserForm to the UserForms collection and then use the Show method of the UserForms collection. Here's an example that assigns the name of a UserForm to the MyForm variable and then displays the UserForm:

```
MyForm = "UserForm1"  
UserForms.Add(MyForm).Show
```

### Loading a UserForm

VBA also has a Load statement. Loading a UserForm loads it into memory, but it's not visible until you use the Show method. To load a UserForm, use a statement like this:

```
Load UserForm1
```

If you have a complex UserForm, you might want to load it into memory before it's needed so that it will appear more quickly when you use the Show method. In the majority of situations, however, you don't need to use the Load statement.

### About event-handler procedures

After the UserForm is displayed, the user interacts with it — selecting an item from a ListBox, clicking a CommandButton, and so on. In official terminology, the user causes an event to occur. For example, clicking a CommandButton causes the Click event for the CommandButton. You need to write procedures that



## 7.8. Closing a UserForm

execute when these events occur. These procedures are sometimes known as event-handler procedures.

**Notes: Event-handler procedures must be located in the Code window for the UserForm. However, your event-handler procedure can call another procedure that's located in a standard VBA module.**

Your VBA code can change the properties of the controls while the UserForm is displayed (that is, at runtime). For example, you could assign to a ListBox control a procedure that changes the text in a Label when an item is selected. This type of manipulation will become clearer later in this chapter.

To close a UserForm, use the Unload command, as shown in this example:

```
Unload UserForm1
```

Or, if the code is located in the code module for the UserForm, you can use the following:

```
Unload Me
```

In this case, the keyword Me refers to the UserForm. Using Me rather than the UserForm's name eliminates the need to modify your code if you change the name of the UserForm.

Normally, your VBA code should include the Unload command after the UserForm has performed its actions. For example, your UserForm may have a CommandButton that serves as an OK button. Clicking this button executes a macro. One of the statements in the macro will unload the UserForm. The UserForm remains visible on the screen until the macro that contains the Unload statement finishes.

When a UserForm is unloaded, its controls are reset to their original values. In other words, your code won't be able to access the user's choices after the UserForm is unloaded. If the user's choice must be used later on (after the UserForm is unloaded), you need to store the value in a Public variable, declared in a standard VBA module. Or you could store the value in a worksheet cell, or even in the Windows registry.

**Notes: A UserForm is automatically unloaded when the user clicks the Close button (the X in the UserForm title bar). This action also triggers a UserForm QueryClose event, followed by a UserForm Terminate event.**

UserForms also have a Hide method. When you invoke this method, the UserForm disappears, but it remains loaded in memory, so your code can still access the various properties of the controls. Here's an example of a statement that hides a UserForm:

```
UserForm1.Hide
```

Or, if the code is in the code module for the UserForm, you can use the following:

```
Me.Hide
```

If for some reason you'd like your UserForm to disappear immediately while its macro is executing, use the Hide method at the top of the procedure. For example, in the following procedure, the UserForm disappears immediately when CommandButton1 is clicked. The last statement in the procedure unloads the UserForm.

```
Private Sub CommandButton1_Click()
```



## 7.9. Creating a UserForm: An Example

```

Me.Hide

Application.ScreenUpdating = True

For r = 1 To 10000
    Cells(r, 1) = r
Next r

Unload Me

End Sub

```

In this example, I set ScreenUpdating to True to force Excel to hide the UserForm completely. Without that statement, the UserForm may actually remain visible.

If you've never created a UserForm, you might want to walk through the example in this section. The example includes step-by-step instructions for creating a simple dialog box and developing a VBA procedure to support the dialog box.

This example uses a UserForm to obtain two pieces of information: a person's name and sex. The dialog box uses a TextBox control to get the name and three OptionButtons to get the sex (Male, Female, or Unknown). The information collected in the dialog box is then sent to the next blank row in a worksheet.

### Creating the UserForm

Figure 13-9 shows the completed UserForm for this example. For best results, start with a new workbook with only one worksheet in it. Then follow these steps:

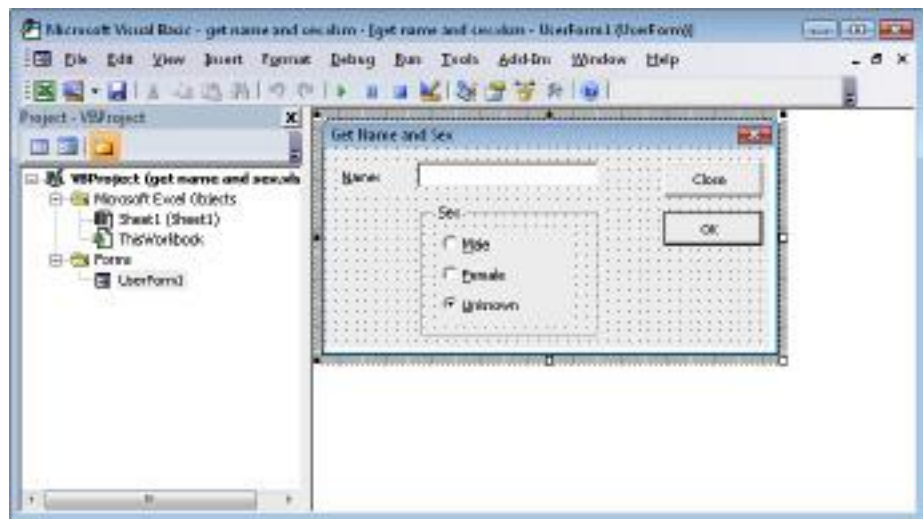


FIGURE 13-9: This dialog box asks the user to enter a name and a sex.

1. Press Alt+F11 to activate the VBE.
2. In the Project window, select the workbook's project and choose Insert⇒UserForm to add an empty UserForm.  
The UserForm's Caption property will have its default value: UserForm1.
3. Use the Properties window to change the UserForm's Caption property to Get Name and Sex.  
(If the Properties window isn't visible, press F4.)
4. Add a Label control and adjust the properties as follows:

Property	Value
Accelerator	N
Caption	Name:
TabIndex	0

5. Add a TextBox control and adjust the properties as follows:

Property	Value
Name	TextName
TabIndex	1

6. Add a Frame control and adjust the properties as follows:

Property	Value
Caption	Sex
TabIndex	2

7. Add an OptionButton control inside the frame and adjust the properties as follows:

Property	Value
Accelerator	M
Caption	Male
Name	OptionMale
TabIndex	0

8. Add another OptionButton control inside the frame and adjust the properties as follows:

Property	Value
Accelerator	F
Caption	Female
Name	OptionFemale
TabIndex	1

9. Add yet another OptionButton control inside the Frame and adjust the properties as follows:

Property	Value
Accelerator	U

Caption	Unknown
Name	OptionUnknown
TabIndex	2
Value	True

10. Add a CommandButton control outside the Frame and adjust the properties as follows:

Property	Value
Caption	OK
Default	True
Name	OKButton
TabIndex	3

11. Add another CommandButton control and adjust the properties as follows:

Property	Value
Caption	Close
Cancel	True
Name	CloseButton
TabIndex	4

### Writing code to display the dialog box

Next, you add an ActiveX CommandButton to the worksheet. This button will execute a procedure that displays the UserForm. Here's how:

1. Activate Excel.  
(Alt+F11 is the shortcut key combination.)
2. Choose Developer⇒Controls⇒Insert and click CommandButton from the ActiveX Controls section.
3. Drag in the worksheet to create the button.

If you like, you can change the caption for the worksheet CommandButton. To do so, right-click the button and choose CommandButton Object⇒Edit from the shortcut menu. You can then edit the text that appears on the CommandButton. To change other properties of the object, right-click and choose Properties. Then make the changes in the Properties box.

4. Double-click the CommandButton.

This step activates the VBE. More specifically, the code module for the worksheet will be displayed, with an empty event-handler procedure for the worksheet's CommandButton.

5. Enter a single statement in the CommandButton1\_Click procedure (see Figure 13-10).

This short procedure uses the Show method of an object (UserForm1) to display the UserForm.

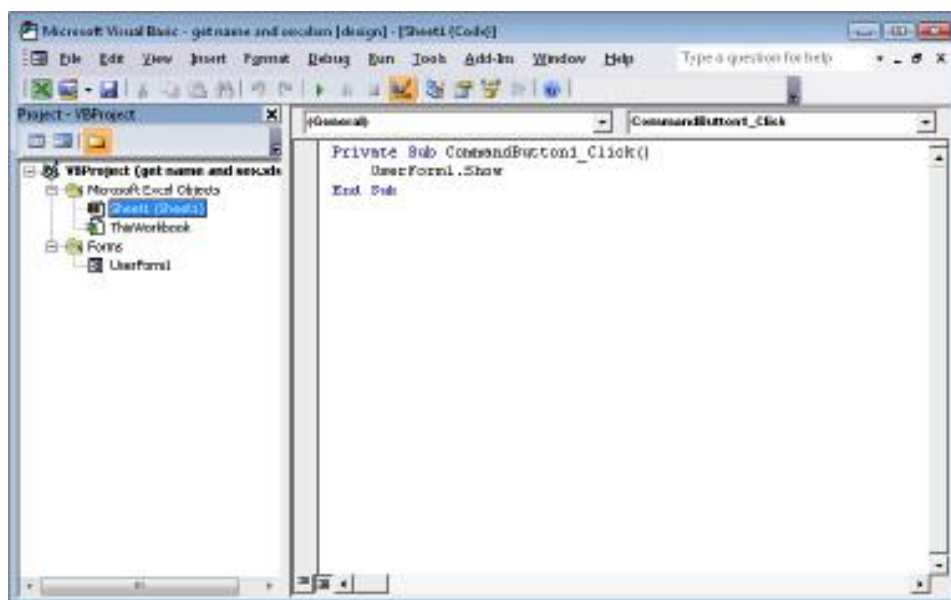


FIGURE 13-10: The CommandButton1\_Click procedure is executed when the button on the worksheet is clicked.

### Testing the dialog box

The next step is to re-activate Excel and try out the procedure that displays the dialog box.

When you exit design mode, clicking the button will display the UserForm (see Figure 13-11).

When the dialog box is displayed, enter some text into the text box and click OK. Nothing happens — which is understandable because you haven't yet created an event-handler procedure for the OK button.

**Notes; Click the X (Close) button in the UserForm title bar to dismiss the dialog box.**

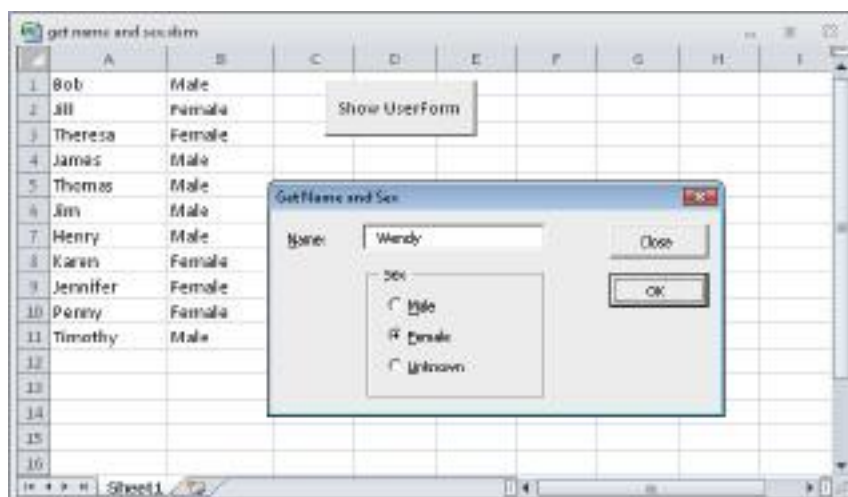


FIGURE 13-11: The CommandButton's Click event procedure displays the UserForm.

**Adding event-handler procedures**

In this section, I explain how to write the procedures that will handle the events that occur when the UserForm is displayed. To continue the example, do the following:

1. Press Alt+F11 to activate the VBE.
2. Make sure that the UserForm is displayed and double-click the CommandButton captioned Close.

This step activates the Code window for the UserForm and inserts an empty procedure named CloseButton\_Click. Notice that this procedure consists of the object's name, an underscore character, and the event that it handles.

3. Modify the procedure as follows.

(This is the event handler for the CloseButton's Click event.)

```
Private Sub CloseButton_Click()  
    Unload UserForm1  
End Sub
```

This procedure, which is executed when the user clicks the Close button, simply unloads the UserForm.

4. Press Shift+F7 to redisplay UserForm1 (or click the View Object icon at the top of the Project Explorer window).
5. Double-click the OK button and enter the following procedure.

(This is the event handler for the OKButton's Click event.)

```
Private Sub OKButton_Click()  
    Dim NextRow As Long  
    ' Make sure Sheet1 is active  
    Sheets("Sheet1").Activate  
    ' Determine the next empty row  
    NextRow = _  
        Application.WorksheetFunction.CountA(Range("A:A")) + 1  
    ' Transfer the name  
    Cells(NextRow, 1) = TextName.Text  
    ' Transfer the sex  
    If OptionMale Then Cells(NextRow, 2) = "Male"  
    If OptionFemale Then Cells(NextRow, 2) = "Female"  
    If OptionUnknown Then Cells(NextRow, 2) = "Unknown"  
    ' Clear the controls for the next entry  
    TextName.Text = ""  
    OptionUnknown = True  
    TextName.SetFocus  
End Sub
```

6. Activate Excel and click the CommandButton again to display the UserForm and then re-run the procedure again.

You'll find that the UserForm controls now function correctly. You can use them to add new names to the list in the worksheet.

Here's how the OKButton\_Click procedure works: First, the procedure makes sure that the proper worksheet (Sheet1) is active. It then uses Excel's COUNTA function to determine the next blank cell in column A. Next, it transfers the text from the TextBox control to column A. It then uses a series of If statements to determine which OptionButton was selected and writes the appropriate text (Male, Female, or Unknown) to column B. Finally, the dialog box is reset to make it ready for the next entry. Notice that clicking OK doesn't close the dialog box. To end data entry (and unload the UserForm), click the Close button.

### Validating the data

Play around with this example some more, and you'll find that it has a small problem: It doesn't ensure that the user actually enters a name into the text box. To make sure that the user enters a name, insert the following code in the OKButton\_Click procedure, before the text is transferred to the worksheet. It ensures that the user enters a name (well, at least some text) in the TextBox. If the TextBox is empty, a message appears, and the focus is set to the TextBox so that the user can try again. The Exit Sub statement ends the procedure with no further action.

```
' Make sure a name is entered  
If TextName.Text = "" Then  
MsgBox "You must enter a name."  
TextName.SetFocus  
Exit Sub  
End If
```

### The finished dialog box

After making all these modifications, you'll find that the dialog box works flawlessly. (Don't forget to test the hot keys.) In real life, you'd probably need to collect more information than just name and sex. However, the same basic principles apply. You just need to deal with more UserForm controls.

## 7.10. Understanding UserForm Events

Each UserForm control (as well as the UserForm itself) is designed to respond to certain types of events, and a user or Excel can trigger these events. For example, clicking a CommandButton generates a Click event for the CommandButton. You can write code that is executed when a particular event occurs.

Some actions generate multiple events. For example, clicking the upward arrow of a SpinButton control generates a SpinUp event and also a Change event. When a UserForm is displayed by using the Show method, Excel generates an Initialize event and an Activate event for the UserForm. (Actually, the Initialize event occurs when the UserForm is loaded into memory and before it's actually displayed.)

### Learning about events

To find out which events are supported by a particular control, do the following:

1. Add a control to a UserForm.

2. Double-click the control to activate the code module for the UserForm.

The VBE will insert an empty event-handler procedure for the default event for the control.

3. Click the drop-down list in the upper-right corner of the module window, and you'll see a complete list of events for the control.

Figure 13-12 shows the list of events for a CheckBox control.

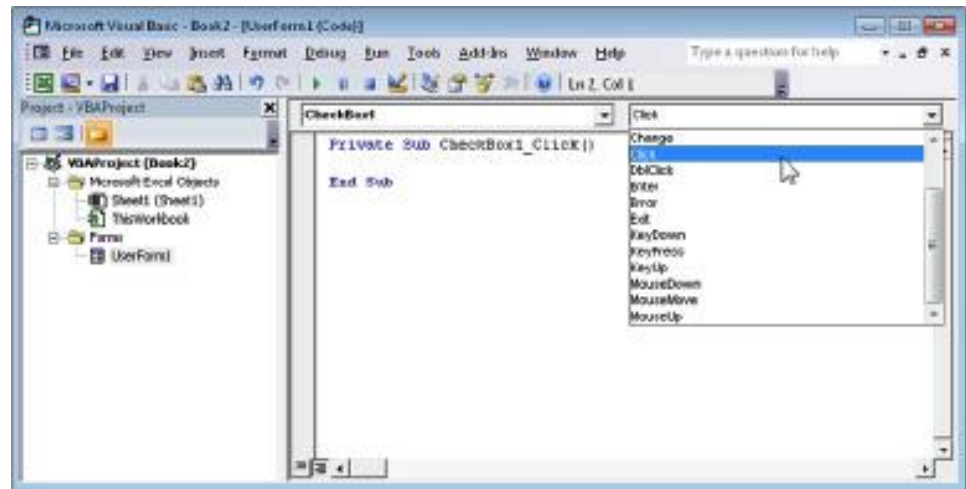


FIGURE 13-12: The event list for a CheckBox control.

4. Select an event from the list, and the VBE will create an empty event-handler procedure for you.

**Notes:** To find out specific details about an event, consult the Help system. The Help system also lists the events available for each control. When you locate an event for an object, make sure that the Help system table of contents is displayed. Then you can see a list of all other events for the object.

**Notes:** Event-handler procedures incorporate the name of the object in the procedure's name. Therefore, if you change the name of a control, you'll also need to make the appropriate changes to the control's event-handler procedure(s). The name changes aren't performed automatically! To make things easy on yourself, it's a good idea to provide names for your controls before you begin creating event-handler procedures.

### UserForm events

Several events are associated with showing and unloading a UserForm:

- Initialize: Occurs before a UserForm is loaded or shown but doesn't occur if the UserForm was previously hidden.
- Activate: Occurs when a UserForm is shown.
- Deactivate: Occurs when a UserForm is deactivated but doesn't occur if the form is hidden.
- QueryClose: Occurs before a UserForm is unloaded.
- Terminate: Occurs after the UserForm is unloaded.

**Notes:** Often, it's critical that you choose the appropriate event for your event-handler procedure and that you understand the order in which the

events occur. Using the Show method invokes the Initialize and Activate events (in that order). Using the Load command invokes only the Initialize event. Using the Unload command triggers the QueryClose and Terminate events (in that order). Using the Hide method doesn't trigger either of these events.

### SpinButton events

To help clarify the concept of events, this section takes a close look at the events associated with a SpinButton control. Some of these events are associated with other controls, and some are unique to the SpinButton control.

Table 13-1 lists all the events for the SpinButton control.

**Table 13-1: SpinButton Events**

Event	Description
AfterUpdate	Occurs after the control is changed through the user interface.
BeforeDragOver	Occurs when a drag-and-drop operation is in progress.
BeforeDropOrPaste	Occurs when the user is about to drop or paste data onto the control.
BeforeUpdate	Occurs before the control is changed.
Change	Occurs when the Value property changes.
Enter	Occurs before the control actually receives the focus from a control on the same UserForm.
Error	Occurs when the control detects an error and can't return the error information to a calling program.
Exit	Occurs immediately before a control loses the focus to another control on the same form.
KeyDown	Occurs when the user presses a key and the object has the focus.
KeyPress	Occurs when the user presses any key that produces a typeable character.
KeyUp	Occurs when the user releases a key and the object has the focus.
SpinDown	Occurs when the user clicks the lower (or left) SpinButton arrow.
SpinUp	Occurs when the user clicks the upper (or right) SpinButton arrow.

A user can operate a SpinButton control by clicking it with the mouse or (if the control has the focus) by using the up-arrow and down-arrow keys.

### Mouse-initiated events

When the user clicks the upper SpinButton arrow, the following events occur in this precise order:



1. Enter (triggered only if the SpinButton did not already have the focus)
2. Change
3. SpinUp

### Keyboard-initiated events

The user can also press Tab to set the focus to the SpinButton and then use the arrow keys to increment or decrement the control. If so, the following events occur (in this order):

1. Enter
2. KeyDown
3. Change
4. SpinUp (or SpinDown)
5. KeyUp

### What about changes via code?

The SpinButton control can also be changed by VBA code — which also triggers the appropriate event(s). For example, the following statement sets the SpinButton1 Value property to 0 and also triggers the Change event for the SpinButton control — but only if the SpinButton value was not already 0:

```
SpinButton1.Value = 0
```

You might think that you could disable events by setting the EnableEvents property of the Application object to False. Unfortunately, this property applies only to events that involve true Excel objects: Workbooks, Worksheets, and Charts.

### Pairing a SpinButton with a TextBox

A SpinButton has a Value property, but this control doesn't have a caption in which to display its value. In many cases, however, you'll want the user to see the SpinButton value. And sometimes you'll want the user to be able to change the SpinButton value directly instead of clicking the SpinButton repeatedly.

The solution is to pair a SpinButton with a TextBox, which enables the user to specify a value either by typing it into the TextBox directly or by clicking the SpinButton to increment or decrement the value in the TextBox.

Figure 13-13 shows a simple example. The SpinButton's Min property is 1, and its Max property is 100. Therefore, clicking the SpinButton's arrows will change its value to an integer between 1 and 100.



FIGURE 13-13: This SpinButton is paired with a TextBox.

The code required to link a SpinButton with a TextBox is relatively simple. It's basically a matter of writing event-handler procedures to ensure that the SpinButton's Value property is always in sync with the TextBox's Text property.

The following procedure is executed whenever the SpinButton's Change event is triggered. That is, the procedure is executed when the user clicks the SpinButton or changes its value by pressing the up arrow or down arrow.

```
Private Sub SpinButton1_Change()  
    TextBox1.Text = SpinButton1.Value  
End Sub
```

The procedure simply assigns the SpinButton's Value to the Text property of the TextBox control. Here, the controls have their default names (SpinButton1 and TextBox1). If the user enters a value directly into the TextBox, its Change event is triggered, and the following procedure is executed:

```
Private Sub TextBox1_Change()  
    NewVal = Val(TextBox1.Text)  
    If NewVal >= SpinButton1.Min And _  
    NewVal <= SpinButton1.Max Then _  
    SpinButton1.Value = NewVal  
End Sub
```

This procedure starts by using VBA's Val function to convert the text in the TextBox to a value. (If the TextBox contains non-numeric text, the Val function returns 0.) The next statement determines whether the value is within the proper range for the SpinButton. If so, the SpinButton's Value property is set to the value entered in the TextBox.

### About the Tag property

Every UserForm and control has a Tag property. This property doesn't represent anything specific, and, by default, is empty. You can use the Tag property to store information for your own use.

For example, you may have a series of TextBox controls in a UserForm. The user may be required to enter text into some but not all of them. You can use the Tag property to identify (for your own use) which fields are required. In this case, you can set the Tag property to a string such as Required. Then when you write code to validate the user's entries, you can refer to the Tag property.

The following example is a function that examines all TextBox controls on UserForm1 and returns the number of required TextBox controls that are empty:

```
Function EmptyCount()  
    Dim ctl As Control  
    EmptyCount = 0  
    For Each ctl In UserForm1.Controls  
        If TypeName(ctl) = "TextBox" Then  
            If ctl.Tag = "Required" Then  
                If ctl.Text = "" Then  
                    EmptyCount = EmptyCount + 1  
                End If  
            End If  
        End If  
    Next ctl  
End Function
```

As you work with UserForms, you'll probably think of other uses for the Tag property.

The example is set up so that clicking the OK button (which is named OKButton) transfers the SpinButton's value to the active cell. The event handler for this CommandButton's Click event is as follows:

```
Private Sub OKButton_Click()  
    ' Enter the value into the active cell  
    If CStr(SpinButton1.Value) = TextBox1.Text Then  
        ActiveCell = SpinButton1.Value  
        Unload Me  
    Else  
        MsgBox "Invalid entry.", vbCritical  
        TextBox1.SetFocus  
        TextBox1.SelStart = 0  
        TextBox1.SelLength = Len(TextBox1.Text)  
    End If  
End Sub
```

This procedure does one final check: It makes sure that the text entered in the TextBox matches the SpinButton's value. This check is necessary in the case of an invalid entry. For example, if the user enters 3r into the TextBox, the SpinButton's value would not be changed, and the result placed in the active cell would not be what the user intended. Notice that the SpinButton's Value property is converted to a string by using the CStr function. This conversion ensures that the comparison won't generate an error if a value is compared with text. If the SpinButton's value doesn't match the TextBox's contents, a message box is displayed. Notice that the focus is set to the TextBox object, and the contents are selected (by using the SelStart and SelLength properties). This setup makes it very easy for the user to correct the entry.

### 7.11. Referencing UserForm Controls

When working with controls on a UserForm, the VBA code is usually contained in the code window for the UserForm. You can also refer to UserForm controls from a general VBA module. To do so, you need to qualify the reference to the control by specifying the UserForm name. For example, consider the following procedure, which is located in a VBA module. It simply displays the UserForm named UserForm1.

```
Sub GetData()  
    UserForm1.Show  
End Sub
```

Assume that UserForm1 contains a text box (named TextBox1), and you want to provide a default value for the text box. You could modify the procedure as follows:

```
Sub GetData()  
    UserForm1.TextBox1.Value = "John Doe"  
    UserForm1.Show  
End Sub
```

Another way to set the default value is to take advantage of the UserForm's Initialize event. You can write code in the UserForm\_Initialize procedure, which is located in the code module for the UserForm. Here's an example:

```
Private Sub UserForm_Initialize()
```

```
TextBox1.Value = "John Doe"
```

```
End Sub
```

Notice that when the control is referenced in the code module for the UserForm, you don't need to qualify the references with the UserForm name. However, qualifying references to controls does have an advantage: You'll then be able to take advantage of the Auto List Members feature, which lets you choose the control names from a drop-down list.

### Understanding the controls collection

The controls on a UserForm make up a collection. For example, the following statement displays the number of controls on UserForm1:

```
MsgBox UserForm1.Controls.Count
```

VBA does not maintain a collection of each control type. For example, there is no collection of CommandButton controls. However, you can determine the type of control by using the TypeName function. The following procedure uses a For Each structure to loop through the Controls collection and then displays the number of CommandButton controls on UserForm1:

```
Sub CountButtons()
```

```
Dim cbCount As Integer
```

```
Dim ctl as Control
```

```
cbCount = 0
```

```
For Each ctl In UserForm1.Controls
```

```
If TypeName(ctl) = "CommandButton" Then _
```

```
cbCount = cbCount + 1
```

```
Next ctl
```

```
MsgBox cbCount
```

```
End Sub
```

### 7.12. A UserForm Checklist

Before you unleash a UserForm on end users, be sure that everything is working correctly. The following checklist should help you identify potential problems:

- Are similar controls the same size?
- Are the controls evenly spaced?
- Is the dialog box too overwhelming? If so, you may want to group the controls by using a MultiPage control.
- Can every control be accessed with a hot key?
- Are any of the hot keys duplicated?
- Is the tab order set correctly?
- Will your VBA code take appropriate action if the user presses Esc or clicks the Close button on the UserForm?
- Are there any misspellings in the text?
- Does the dialog box have an appropriate caption?
- Will the dialog box display properly at all video resolutions?
- Are the controls grouped logically (by function)?
- Do ScrollBar and SpinButton controls allow valid values only?

- Does the UserForm use any controls that might not be installed on every system?
- Are ListBoxes set properly (Single, Multi, or Extended)? See Chapter 14 for details on ListBox controls.

## 8. Working with Pivot Tables

### 8.1. An Introductory Pivot Table Example

Excel's pivot table feature is, arguably, its most innovative and powerful feature. Pivot tables first appeared in Excel 5, and the feature has been improved in every subsequent version. This chapter is not an introduction to pivot tables. I assume that you're familiar with this feature and its terminology and that you know how to create and modify pivot tables manually.

As you probably know, creating a pivot table from a database or list enables you to summarize data in ways that otherwise would not be possible — and it's amazingly fast and requires no formulas. You also can write VBA code to generate and modify pivot tables.

This section gets the ball rolling with a simple example of using VBA to create a pivot table.

Figure 17-1 shows a very simple worksheet range. It contains four fields: SalesRep, Region, Month, and Sales. Each record describes the sales for a particular sales representative in a particular month.

	A	B	C	D	E
1	SalesRep	Region	Month	Sales	
2	Amy	North	Jan	33,488	
3	Amy	North	Feb	47,008	
4	Amy	North	Mar	32,128	
5	Bob	North	Jan	34,736	
6	Bob	North	Feb	52,872	
7	Bob	North	Mar	76,128	
8	Chuck	South	Jan	41,536	
9	Chuck	South	Feb	23,192	
10	Chuck	South	Mar	21,736	
11	Doug	South	Jan	44,834	
12	Doug	South	Feb	32,002	
13	Doug	South	Mar	23,932	
14					
15					
16					

FIGURE 17-1: This table is a good candidate for a pivot table.

#### Creating a pivot table

Figure 17-2 shows a pivot table created from the data, along with the PivotTable Field List task bar. This pivot table summarizes the sales performance by sales representative and month. This pivot table is set up with the following fields:

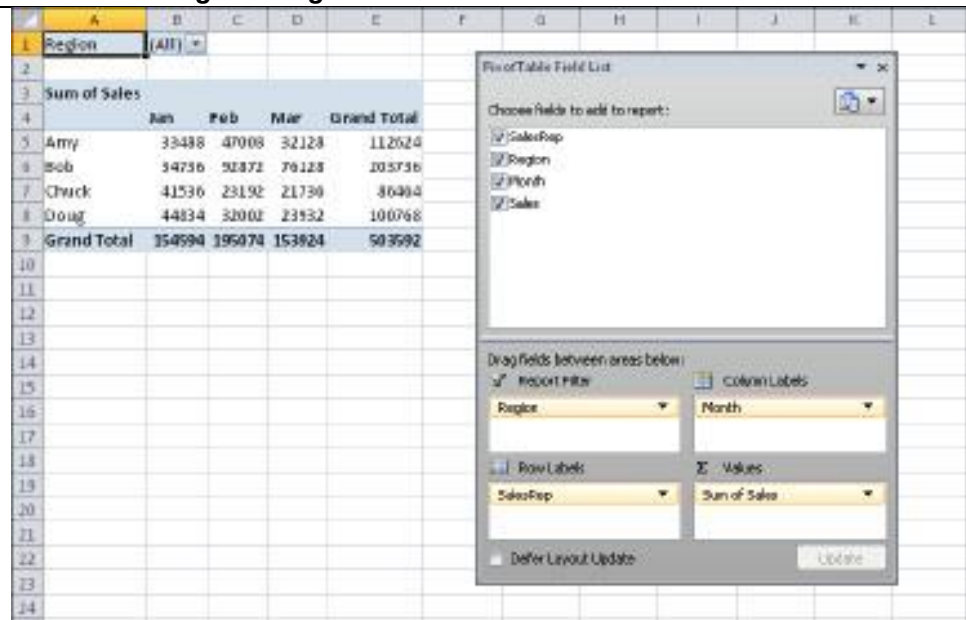


FIGURE 17-2: A pivot table created from the data in Figure 17-1.

- **Region:** A report filter field in the pivot table.
- **SalesRep:** A row field in the pivot table.
- **Month:** A column field in the pivot table.
- **Sales:** A values field in the pivot table that uses the Sum function.

I turned on the macro recorder before I created this pivot table and specified a new worksheet for the pivot table location. The code that was generated follows:

```
Sub RecordedMacro()
Range("A1").Select
Sheets.Add
ActiveWorkbook.PivotCaches.Create _
(SourceType:=xlDatabase, _
SourceData:="Sheet1!R1C1:R13C4", _
Version:=xlPivotTableVersion14).CreatePivotTable _
TableDestination:="Sheet2!R3C1", _
TableName:="PivotTable1", _
DefaultVersion:=xlPivotTableVersion14)
Sheets("Sheet2").Select
Cells(3, 1).Select
With ActiveSheet.PivotTables("PivotTable1") _
.PivotFields("SalesRep")
.Orientation = xlRowField
.Position = 1
End With
With ActiveSheet.PivotTables("PivotTable1") _
.PivotFields("Month")
```

```
.Orientation = xlColumnField  
.Position = 1  
End With  
ActiveSheet.PivotTables("PivotTable1") _  
.AddDataField ActiveSheet.PivotTables("PivotTable1") _  
.PivotFields("Sales"), "Sum of Sales", xlSum  
With ActiveSheet.PivotTables("PivotTable1"). _  
PivotFields("Region")  
.Orientation = xlPageField  
.Position = 1  
End With  
End Sub
```

If you execute this macro, it will almost certainly produce an error. Examine the code, and you'll see that the macro recorder hard-coded the worksheet name (Sheet2) for the pivot table. If that sheet already exists (or if the new sheet that's added has a different name), the macro ends with an error. It also hard-coded the pivot table name. The name won't be PivotTable1 if the workbook has other pivot tables.

But even though the recorded macro doesn't work, it's not completely useless. The code provides lots of insight for writing code to generate pivot tables.

### Data appropriate for a pivot table

A pivot table requires that your data is in the form of a rectangular database. You can store the database in either a worksheet range (which can be a table or just a normal range) or an external database file. Although Excel can generate a pivot table from any database, not all databases benefit.

Generally speaking, fields in a database table consist of two types:

- **Data:** Contains a value or data to be summarized. For the bank account example, the Amount field is a data field.
- **Category:** Describes the data. For the bank account data, the Date, AcctType, OpenedBy, Branch, and Customer fields are category fields because they describe the data in the Amount field.

A database table that's appropriate for a pivot table is said to be normalized. In other words, each record (or row) contains information that describes the data.

A single database table can have any number of data fields and category fields. When you create a pivot table, you usually want to summarize one or more of the data fields. Conversely, the values in the category fields appear in the pivot table as rows, columns, or filters.

If you're not clear on the concept, the companion CD-ROM contains a workbook named normalized data.xlsx. This workbook contains an example of a range of data before and after being normalized so it's suitable for a pivot table.

### Examining the recorded code for the pivot table

VBA code that works with pivot tables can be confusing. To make any sense of the recorded macro, you need to know about a few relevant objects, all of which are explained in the Help system.



- **PivotCaches:** A collection of PivotCache objects in a Workbook object (the data used by a pivot table is stored in a pivot cache).
- **PivotTables:** A collection of PivotTable objects in a Worksheet object.
- **PivotFields:** A collection of fields in a PivotTable object.
- **PivotItems:** A collection of individual data items within a field category.
- **CreatePivotTable:** A method that creates a pivot table by using the data in a pivot cache.

### **Cleaning up the recorded pivot table code**

As with most recorded macros, the preceding example isn't as efficient as it could be. And, as I noted, it's very likely to generate an error. You can simplify the code to make it more understandable and also to prevent the error. The hand-crafted code that follows generates the same pivot table as the procedure previously listed:

```
Sub CreatePivotTable()  
    Dim PTCache As PivotCache  
    Dim PT As PivotTable  
    ' Create the cache  
    Set PTCache = ActiveWorkbook.PivotCaches.Create( _  
        SourceType:=xlDatabase, _  
        SourceData:=Range("A1").CurrentRegion)  
    ' Add a new sheet for the pivot table  
    Worksheets.Add  
    ' Create the pivot table  
    Set PT = ActiveSheet.PivotTables.Add( _  
        PivotCache:=PTCache, _  
        TableDestination:=Range("A3"))  
    ' Specify the fields  
    With PT  
        .PivotFields("Region").Orientation = xlPageField  
        .PivotFields("Month").Orientation = xlColumnField  
        .PivotFields("SalesRep").Orientation = xlRowField  
        .PivotFields("Sales").Orientation = xlDataField  
        'no field captions  
        .DisplayFieldCaptions = False  
    End With  
End Sub
```

The CreatePivotTable procedure is simplified (and might be easier to understand) because it declares two object variables: PTCache and PT. A new PivotCache object is created by using the Create method. A worksheet is added, and it becomes the active sheet (the destination for the pivot table). Then a new PivotTable object is created by using the Add method of the PivotTables

collection. The last section of the code adds the four fields to the pivot table and specifies their location within it by assigning a value to the Orientation property.

The original macro hard-coded both the data range used to create the PivotCache object ('Sheet1!R1C1:R13C4') and the pivot table location (Sheet2). In the CreatePivotTable procedure, the pivot table is based on the current region surrounding cell A1. This ensures that the macro will continue to work properly if more data is added.

Adding the worksheet before the pivot table is created eliminates the need to hard-code the sheet reference. Yet another difference is that the hand-written macro doesn't specify a pivot table name. Because the PT object variable is created, your code doesn't ever have to refer to the pivot table by name.

### Pivot table compatibility

If you plan to share a workbook that contains a pivot table with users of previous versions of Excel, you need to pay careful attention to compatibility. If you look at the recorded macro in the "Creating a pivot table" section, you see the following statement:

*DefaultVersion:=xlPivotTableVersion14*

If your workbook is in compatibility mode, the recorded statement is:

*DefaultVersion:=xlPivotTableVersion10*

You'll also find that the recorded code is completely different because Microsoft has made significant changes in pivot tables beginning with Excel 2007.

Assume that you create a pivot table in Excel 2010 and give the workbook to a coworker who has Excel 2003. The coworker will see the pivot table, but it will not be refreshable. In other words, it's just a dead table of numbers.

To create a backward compatible pivot table in Excel 2010, you must save your file in XLS format and then re-open it. After doing so, pivot tables that you create will work with versions prior to Excel 2007. But, of course, you won't be able to take advantage of all the new pivot table features introduced in Excel 2007 and Excel 2010.

Fortunately, Excel's Compatibility Checker will alert you regarding this type of compatibility issue (see the accompanying figure). However, it won't check your pivot table-related macros for compatibility.



The macros in this chapter do not generate backward compatible pivot tables.

## 8.2. Creating a More Complex Pivot Table

**Notes:** The code also could be more general through the use of indices rather than literal strings for the PivotFields collections. This way, if the user changes the column headings, the code will still work. For example, more general code would use PivotFields(1) rather than PivotFields("Region").

As always, the best way to master this topic is to record your actions within a macro to find out its relevant objects, methods, and properties. Then study the Help topics to understand how everything fits together. In almost every case, you'll need to modify the recorded macros. Or, after you understand how to work with pivot tables, you can write code from scratch and avoid the macro recorder.

In this section, I present VBA code to create a relatively complex pivot table.

Figure 17-3 shows part of a large worksheet table. This table has 15,840 rows and consists of hierarchical budget data for a corporation. The corporation has five divisions, and each division contains 11 departments. Each department has four budget categories, and each budget category contains several budget items. Budgeted and actual amounts are included for each of the 12 months. The goal is to summarize this information with a pivot table.

	A	B	C	D	E	F	G
1	Division	Department	Category	Item	Month	Budget	Actual
2	N. America	DataProcessing	Compensation	Salaries	Jan	2513	3165
3	N. America	DataProcessing	Compensation	Benefits	Jan	4456	2980
4	N. America	DataProcessing	Compensation	Bonuses	Jan	3718	3029
5	N. America	DataProcessing	Compensation	Commissions	Jan	3133	2815
6	N. America	DataProcessing	Compensation	Payroll Taxes	Jan	3559	3770
7	N. America	DataProcessing	Compensation	Training	Jan	3099	3559
8	N. America	DataProcessing	Compensation	Conferences	Jan	2311	3159
9	N. America	DataProcessing	Compensation	Entertainment	Jan	2632	2633
10	N. America	DataProcessing	Facility	Rent	Jan	2835	2905
11	N. America	DataProcessing	Facility	Lease	Jan	3450	2631
12	N. America	DataProcessing	Facility	Utilities	Jan	4111	5098
13	N. America	DataProcessing	Facility	Maintenance	Jan	3070	2870
14	N. America	DataProcessing	Facility	Telephone	Jan	5827	4329
15	N. America	DataProcessing	Facility	Other	Jan	3843	3322
16	N. America	DataProcessing	Supplies & Services	General Office	Jan	2842	5215
17	N. America	DataProcessing	Supplies & Services	Computer Supplies	Jan	3052	4098
18	N. America	DataProcessing	Supplies & Services	Books & Subs	Jan	4546	5361
19	N. America	DataProcessing	Supplies & Services	Outside Services	Jan	2849	3717
20	N. America	DataProcessing	Supplies & Services	Other	Jan	3328	3116
21	N. America	DataProcessing	Equipment	Computer Hardware	Jan	3088	2728
22	N. America	DataProcessing	Equipment	Software	Jan	4226	2675
23	N. America	DataProcessing	Equipment	Photocopiers	Jan	3780	3514
24	N. America	DataProcessing	Equipment	Telecommunications	Jan	3853	3664
25	N. America	DataProcessing	Equipment	Other	Jan	2851	4380
26	N. America	Human Resources	Compensation	Salaries	Jan	3614	3501
27	N. America	Human Resources	Compensation	Benefits	Jan	2859	4493
28	N. America	Human Resources	Compensation	Bonuses	Jan	3010	2676

FIGURE 17-3: The data in this workbook will be summarized in a pivot table.

Figure 17-4 shows a pivot table created from the data. Notice that the pivot table contains a calculated field named Variance. This field is the difference between the Budget amount and the Actual amount.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Division	(All)												
2	Category	(All)												
3														
4		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Grand Total
5	Accounting													
6	Budget	422,495	433,317	420,522	417,904	411,320	414,012	427,452	410,539	432,134	423,670	428,602	430,445	5,044,919
7	Actual	422,662	433,363	436,522	420,672	431,309	429,999	425,879	415,259	417,461	417,686	425,271	420,826	5,055,951
8	Variance	-9,207	20,354	4,300	-2,768	-19,488	-15,981	1,552	3,277	-5,267	3,672	1,331	-1,581	-11,041
9	Advertising													
10	Budget	424,590	439,351	437,340	420,524	427,150	424,169	421,183	420,245	429,454	412,070	411,896	423,101	5,051,479
11	Actual	418,009	420,820	425,437	417,510	419,996	420,150	420,958	420,258	416,067	418,212	411,799	424,492	5,049,253
12	Variance	6,582	-1,469	-7,435	3,014	7,154	-4,161	-7,775	-6,013	13,387	-7,154	9,397	-1,391	2,227
13	Auto Processing													
14	Budget	422,197	422,657	439,959	417,260	422,848	421,038	421,676	418,069	419,999	418,752	421,106	429,679	5,053,364
15	Actual	414,743	430,890	430,545	424,214	411,775	421,906	420,216	414,064	416,912	424,262	417,470	469,644	5,050,649
16	Variance	7,454	-28,933	-30,386	-6,954	11,073	-9,071	1,468	5,127	0,086	-12,560	3,620	20,935	-6,285
17	Human Resources													
18	Budget	422,053	425,313	435,834	423,036	423,514	419,602	415,197	419,701	422,762	413,741	419,592	422,746	5,037,273
19	Actual	424,594	429,175	467,853	429,187	410,258	421,870	428,551	422,468	422,252	421,889	415,125	417,222	5,050,034
20	Variance	-2,681	-3,862	-11,531	-6,149	13,256	-2,268	-13,354	-2,766	0,510	-8,097	-4,533	5,524	-12,761
21	Information													
22	Budget	413,530	427,575	429,527	422,299	415,290	414,005	413,148	425,207	412,294	414,242	427,521	420,190	5,026,107
23	Actual	415,019	406,592	426,127	418,225	431,307	415,201	416,358	411,338	422,584	418,132	424,041	426,461	5,020,078
24	Variance	-2,299	21,393	-7,300	4,076	-16,009	1,604	-3,201	15,948	-10,290	-1,890	3,480	-6,271	-2,769
25	Public Relations													
26	Budget	424,696	434,507	435,179	417,100	426,223	408,425	422,138	416,145	429,236	418,292	414,600	421,344	5,019,764
27	Actual	413,526	434,604	425,476	414,040	396,672	416,201	420,028	427,948	420,187	406,537	425,407	422,857	4,994,444
28	Variance	11,370	0,423	-9,297	3,060	29,551	-7,776	-1,688	-11,803	6,029	1,745	-18,495	0,191	28,320
29	R&D													
30	Budget	417,771	429,890	424,866	421,539	417,440	421,174	417,152	415,008	417,920	417,732	419,540	429,801	5,037,638
31	Actual	432,037	429,844	429,595	427,967	412,035	425,952	426,668	424,589	411,587	421,449	423,290	429,115	5,079,222
32	Variance	-14,240	3,236	4,471	-6,028	5,405	-4,756	-9,535	-11,268	6,362	-3,667	-3,307	-9,232	-41,584
33	Sales													
34	Budget	429,059	421,962	437,314	420,302	422,409	426,002	426,468	420,079	422,334	428,271	408,725	411,322	5,047,958
35	Actual	435,505	421,251	461,861	433,912	425,620	420,596	424,737	416,838	408,090	417,463	415,400	423,360	5,020,513
36	Variance	-13,996	9,711	-9,155	11,990	-5,211	-1,794	5,729	4,241	14,584	8,808	-6,795	-2,938	27,628
37	Security													
38	Budget	419,195	419,294	413,258	421,700	421,875	421,231	417,361	410,715	417,112	438,013	412,302	429,939	5,024,025
39	Actual	409,496	413,697	427,401	419,221	421,266	420,388	429,328	424,682	416,480	415,699	419,717	423,931	5,020,686
40	Variance	9,709	5,597	-14,143	2,479	6,609	9,043	-6,436	-15,067	-1,280	14,444	1,585	-3,892	-6,661
41	Shipping													
42	Budget	429,595	429,817	425,875	419,617	433,168	421,752	413,338	429,081	428,164	428,851	412,505	425,807	5,094,402
43	Actual	413,039	413,777	434,932	411,949	410,561	422,513	411,461	423,136	420,974	419,450	429,434	426,709	5,006,214
44	Variance	15,760	35,340	-9,343	7,668	22,607	-1,181	1,848	6,699	7,190	9,393	-15,669	6,398	88,188
45	Training													
46	Budget	415,005	422,600	413,129	429,720	417,812	420,315	416,644	427,315	420,713	412,583	418,589	425,366	5,026,379
47	Actual	429,292	419,292	426,344	420,360	433,136	421,086	415,138	420,013	422,749	418,739	431,727	422,896	5,076,113
48	Variance	-7,687	3,308	-13,215	-10,640	-15,324	7,229	1,507	6,502	6,004	-6,296	-13,150	-7,530	-49,734
49	Total Budget	4,632,544	4,668,183	4,609,732	4,616,871	4,615,537	4,621,805	4,615,751	4,619,879	4,646,143	4,606,193	4,604,623	4,614,329	55,449,272
50	Total Actual	4,617,889	4,622,593	4,619,289	4,611,855	4,663,912	4,659,419	4,645,848	4,620,522	4,603,114	4,606,485	4,628,971	4,613,847	55,448,755
51	Total Variance	14,655	45,590	-15,557	5,016	-48,375	5,386	26,903	16,727	43,029	-4,292	-24,348	10,482	51,517
52														

FIGURE 17-4: A pivot table created from the budget data.

**Notes:** Another option is to insert a new column in the table and create a formula to calculate the difference between the budget and actual amounts. If the data is from an external source (rather than in a worksheet), that option may not be possible.

### The code that created the pivot table

Here's the VBA code that created the pivot table:

```
Sub CreatePivotTable()
    Dim PTcache As PivotCache
    Dim PT As PivotTable
    Application.ScreenUpdating = False
    ' Delete PivotSheet if it exists
    On Error Resume Next
    Application.DisplayAlerts = False
    Sheets("PivotSheet").Delete
    On Error GoTo 0
    ' Create a Pivot Cache
```

```

Set PTcache = ActiveWorkbook.PivotCaches.Create( _
SourceTypes:=xlDatabase, _
SourceData:=Range("A1").CurrentRegion.Address)
' Add new worksheet
Worksheets.Add
ActiveSheet.Name = "PivotSheet"
ActiveWindow.DisplayGridlines = False
' Create the Pivot Table from the Cache
Set PT = ActiveSheet.PivotTables.Add( _
PivotCache:=PTcache, _
TableDestination:=Range("A1"), _
TableName:="BudgetPivot")
With PT
' Add fields
.PivotFields("Category").Orientation = xlPageField
.PivotFields("Division").Orientation = xlPageField
.PivotFields("Department").Orientation = xlRowField
.PivotFields("Month").Orientation = xlColumnField
.PivotFields("Budget").Orientation = xlDataField
.PivotFields("Actual").Orientation = xlDataField
.DataPivotField.Orientation = xlRowField
' Add a calculated field to compute variance
.CalculatedFields.Add "Variance", "=Budget-Actual"
.PivotFields("Variance").Orientation = xlDataField
' Specify a number format
.DataBodyRange.NumberFormat = "0,000"
' Apply a style
.TableStyle2 = "PivotStyleMedium2"
' Hide Field Headers
.DisplayFieldCaptions = False
' Change the captions
.PivotFields("Sum of Budget").Caption = " Budget"
.PivotFields("Sum of Actual").Caption = " Actual"
.PivotFields("Sum of Variance").Caption = " Variance"
End With
End Sub

```

### How the more complex pivot table works

The CreatePivotTable procedure starts by deleting the PivotSheet worksheet if it already exists. It then creates a PivotCache object, inserts a new worksheet



named PivotSheet, and creates the pivot table from the PivotCache. The code then adds the following fields to the pivot table:

- **Category:** A report filter (page) field
- **Division:** A report filter (page) field
- **Department:** A row field
- **Month:** A column field
- **Budget:** A data field
- **Actual:** A data field

Notice that the Orientation property of the DataPivotField is set to xlRowField in the following statement:

```
.DataPivotField.Orientation = xlRowField
```

This statement determines the overall orientation of the pivot table, and it represents the Sum Value field in the Pivot Table Field list (see Figure 17-5). Try moving that field to the Column Labels section to see how it affects the pivot table layout.

Next, the procedure uses the Add method of the CalculatedFields collection to create the calculated field Variance, which subtracts the Actual amount from the Budget amount. This calculated field is assigned as a data field.

**Notes:** To add a calculated field to a pivot table manually, use the **PivotTable⇒Options ⇒Calculations⇒Fields, Items, & Sets ⇒Calculated Field** command, which displays the **Insert Calculated Field** dialog box.

Finally, the code makes a few cosmetic adjustments:

- Applies a number format to the DataBodyRange (which represents the entire pivot table data).
- Applies a style.
- Hides the captions (equivalent to the PivotTable Tools⇒Options⇒Show ⇒Field Headers control).

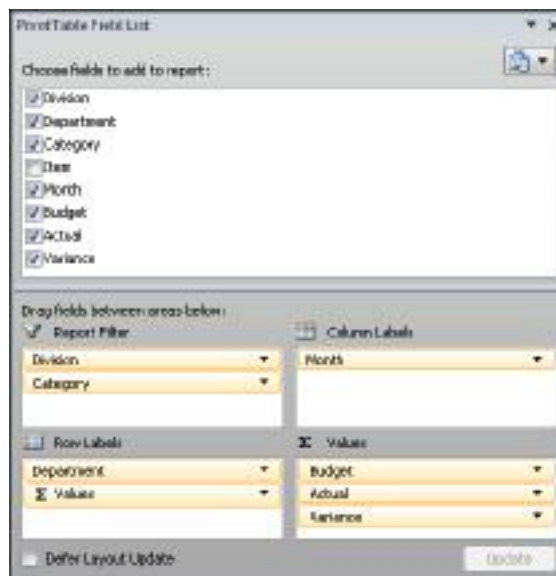


FIGURE 17-5: The Pivot Table Field List.

- Changes the captions displayed in the pivot table. For example, Sum of Budget is replaced by Budget. Note that the string Budget is preceded by a space. Excel doesn't allow you to change a caption that

### 8.3. Creating Multiple Pivot Tables

corresponds to a field name, so adding a space gets around this restriction.

The final example creates a series of pivot tables that summarize data collected in a customer survey. That data is stored in a worksheet database (see Figure 17-6) and consists of 150 rows. Each row contains the respondent's sex plus a numerical rating using a 1–5 scale for each of the 14 survey items.

ItemID	Sex	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Subject1	Male	3	4	4	4	4	4	4	4	4	4	4	4	4	4
Subject2	Female	2	5	1	1	1	1	1	1	1	1	1	1	1	1
Subject3	Male	3	1	4	2	3	3	2	1	2	3	2	4	3	2
Subject4	Male	2	1	3	5	1	2	3	4	2	1	3	4	1	2
Subject5	Female	2	2	5	5	4	2	1	5	5	2	3	4	2	5
Subject6	Female	2	3	3	3	1	1	3	4	2	2	1	2	2	3
Subject7	Female	2	4	5	4	5	3	2	5	4	4	1	5	4	4
Subject8	Male	3	2	1	2	3	4	3	1	2	4	3	4	4	2
Subject9	Female	3	4	4	4	3	3	4	1	4	1	2	1	3	4
Subject10	Male	2	1	5	5	5	1	4	1	2	2	5	1	3	2
Subject11	Male	4	3	3	2	3	2	4	3	1	4	2	2	4	1
Subject12	Female	2	1	4	5	5	5	3	1	4	1	2	3	4	4
Subject13	Female	4	3	4	5	1	5	3	5	2	2	5	1	4	2
Subject14	Female	2	3	4	2	1	1	2	2	1	3	1	1	3	1
Subject15	Female	2	3	5	1	1	2	4	1	3	4	2	5	4	3
Subject16	Male	2	3	1	3	4	3	4	4	3	3	4	1	3	3
Subject17	Female	3	4	3	3	3	4	4	3	2	4	1	1	4	2
Subject18	Male	2	5	5	5	5	3	4	2	3	2	3	3	2	3
Subject19	Female	2	3	3	4	3	3	3	1	1	3	2	3	3	1
Subject20	Male	2	2	5	2	1	5	5	3	1	5	2	4	5	1
Subject21	Male	3	4	1	4	3	3	3	1	4	1	3	3	1	4
Subject22	Male	2	1	3	3	3	2	1	2	2	3	3	2	2	2
Subject23	Male	4	3	4	2	1	2	1	2	1	4	4	1	4	2
Subject24	Female	2	1	2	3	3	3	1	4	1	2	1	1	1	2
Subject25	Female	2	2	4	3	3	5	2	3	2	2	5	2	1	2
Subject26	Male	2	3	4	2	1	3	3	2	1	5	2	1	1	1
Subject27	Male	1	3	1	1	1	2	2	1	3	3	2	1	2	1

FIGURE 17-6: Creating a series of pivot tables will summarize this survey data.

Figure 17-7 shows a few of the 28 pivot tables produced by the macro. Each survey item is summarized in two pivot tables (one showing percentages, and one showing the actual frequencies).

The VBA code that created the pivot tables follows:

```
Sub MakePivotTables()
    ' This procedure creates 28 pivot tables
    Dim PTCache As PivotCache
    Dim PT As PivotTable
    Dim SummarySheet As Worksheet
    Dim ItemName As String
    Dim Row As Long, Col As Long, i As Long
    Application.ScreenUpdating = False
    ' Delete Summary sheet if it exists
    On Error Resume Next
    Application.DisplayAlerts = False
    Sheets("Summary").Delete
    On Error GoTo 0
    ' Add Summary sheet
    Set SummarySheet = Worksheets.Add
    ActiveSheet.Name = "Summary"
    ' Create Pivot Cache
    Set PTCache = ActiveWorkbook.PivotCaches.Create( _
```

SourceType:=xlDatabase, \_

SourceData:=Sheets("SurveyData").Range("A1"). \_

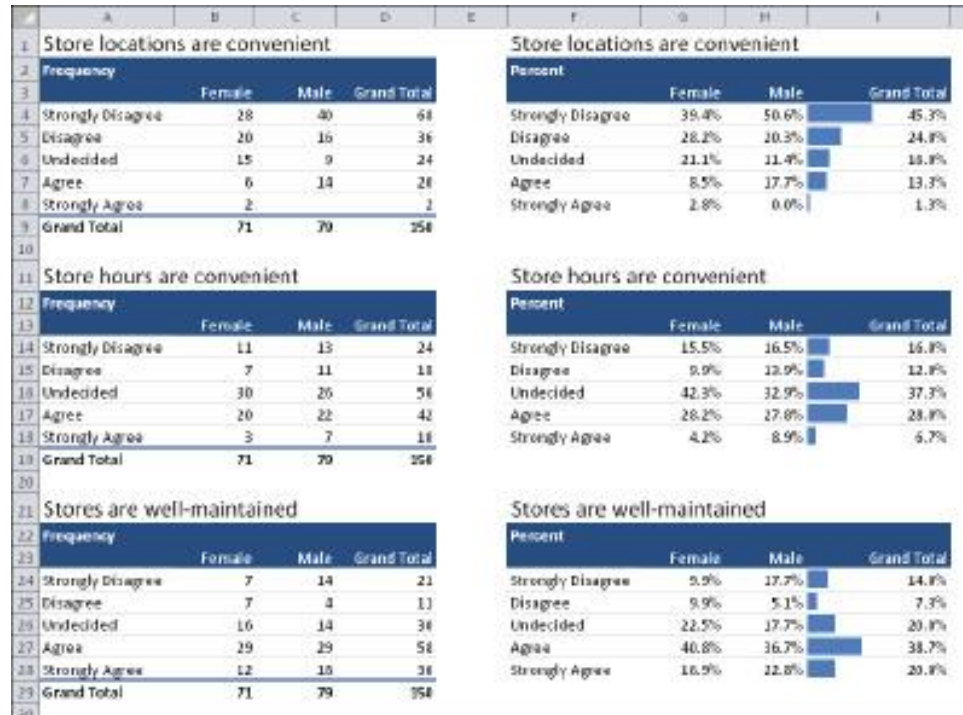


FIGURE 17-7: Six of the 28 pivot tables created by a VBA procedure.

CurrentRegion)

Row = 1

For i = 1 To 14

For Col = 1 To 6 Step 5 '2 columns

ItemName = Sheets("SurveyData").Cells(1, i + 2)

With Cells(Row, Col)

.Value = ItemName

.Font.Size = 16

End With

' Create pivot table

Set PT = ActiveSheet.PivotTables.Add( \_

PivotCache:=PTCache, \_

TableDestination:=SummarySheet.Cells(Row + 1, Col))

' Add the fields

If Col = 1 Then 'Frequency tables

With PT.PivotFields(ItemName)

.Orientation = xlDataField

.Name = "Frequency"

.Function = xlCount

End With



```

Else ' Percent tables
With PT.PivotFields(ItemName)
.Orientation = xlDataField
.Name = "Percent"
.Function = xlCount
.Calculation = xlPercentOfColumn
.NumberFormat = "0.0%"
End With
End If
PT.PivotFields(ItemName).Orientation = xlRowField
PT.PivotFields("Sex").Orientation = xlColumnField
PT.TableStyle2 = "PivotStyleMedium2"
PT.DisplayFieldCaptions = False
If Col = 6 Then
' add data bars to the last column
PT.ColumnGrand = False
PT.DataBodyRange.Columns(3).FormatConditions. _
AddDatabar
With pt.DataBodyRange.Columns(3).FormatConditions(1)
.BarFillType = xlDataBarFillSolid
.MinPoint.Modify newtype:=xlConditionValueNumber, newvalue:=0
.MaxPoint.Modify newtype:=xlConditionValueNumber, newvalue:=1
End With
End If
Next Col
Row = Row + 10
Next i
' Replace numbers with descriptive text
With Range("A:A,F:F")
.Replace "1", "Strongly Disagree"
.Replace "2", "Disagree"
.Replace "3", "Undecided"
.Replace "4", "Agree"
.Replace "5", "Strongly Agree"
End With
End Sub

```

Notice that all these pivot tables were created from a single PivotCache object.

The pivot tables are created within a nested loop. The Col loop counter goes from 1 to 6 by using the Step parameter. The instructions vary a bit for the second column of pivot tables. Specifically, the pivot tables in the second column do the following:

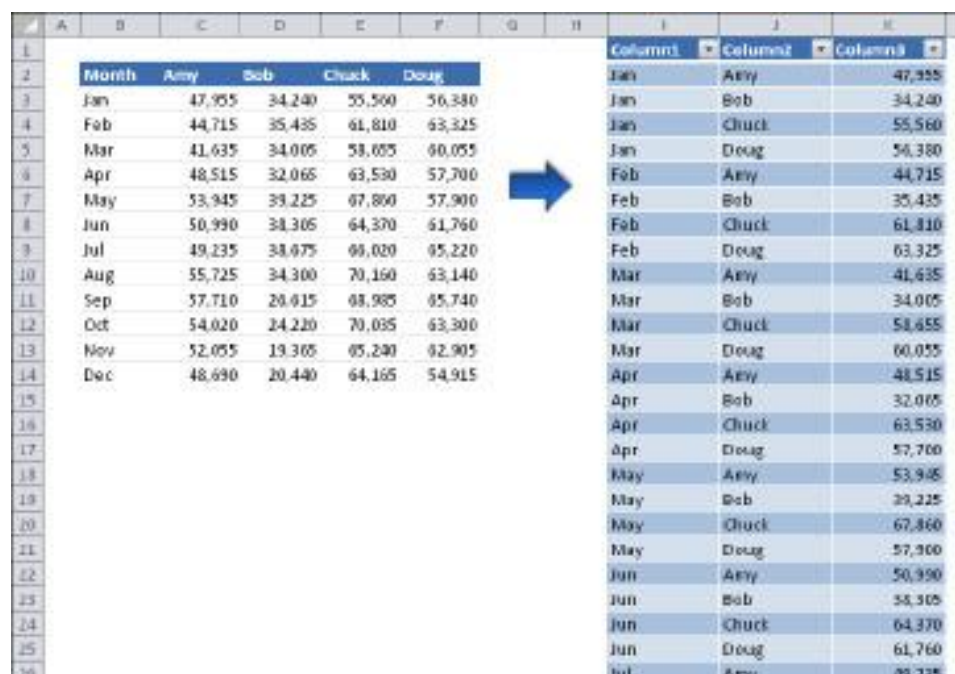
- Display the count as a percent of the column.
- Do not show grand totals for the rows.
- Are assigned a number format.
- Display format conditioning data bars.

The Row variable keeps track of the starting row of each pivot table. The final step is to replace the numeric categories in columns A and F with text. For example, 1 is replaced with Strongly Agree.

#### 8.4. Creating a Reverse Pivot Table

A pivot table is a summary of data in a table. But what if you have a summary table, and you'd like to create a table from it? Figure 17-8 shows an example. Range B2:F14 contains a summary table — similar to a very simple pivot table. Columns I:K contain a 48-row table created from the summary table. In the table, each row contains one data point, and the first two columns describe that data point. In other words, the transformed data is normalized. (See the sidebar, “Data appropriate for a pivot table,” earlier in this chapter.)

Excel doesn't provide a way to transform a summary table into a normalized table, so it's a good job for a VBA macro. After I created this macro, I spent a bit more time and added a UserForm, shown in Figure 17-9. The UserForm gets the input and output ranges and also has an option to convert the output range to a table.



Month	Amy	Bob	Chuck	Doug
Jan	47,955	34,240	55,560	56,380
Feb	44,715	35,435	61,810	63,325
Mar	41,635	34,005	58,055	60,055
Apr	48,515	32,065	63,530	57,700
May	53,945	39,225	67,860	57,900
Jun	50,990	38,305	64,370	61,760
Jul	49,235	38,075	66,020	65,220
Aug	55,725	34,300	70,160	63,140
Sep	57,710	29,615	68,985	65,740
Oct	54,020	24,220	70,035	63,300
Nov	52,055	19,365	65,240	62,905
Dec	48,690	20,440	64,165	54,915

Column1	Column2	Column3
Jan	Amy	47,955
Jan	Bob	34,240
Jan	Chuck	55,560
Jan	Doug	56,380
Feb	Amy	44,715
Feb	Bob	35,435
Feb	Chuck	61,810
Feb	Doug	63,325
Mar	Amy	41,635
Mar	Bob	34,005
Mar	Chuck	58,055
Mar	Doug	60,055
Apr	Amy	48,515
Apr	Bob	32,065
Apr	Chuck	63,530
Apr	Doug	57,700
May	Amy	53,945
May	Bob	39,225
May	Chuck	67,860
May	Doug	57,900
Jun	Amy	50,990
Jun	Bob	38,305
Jun	Chuck	64,370
Jun	Doug	61,760
Jul	Amy	49,235

FIGURE 17-8: The summary table on the left will be converted to the table on the right.

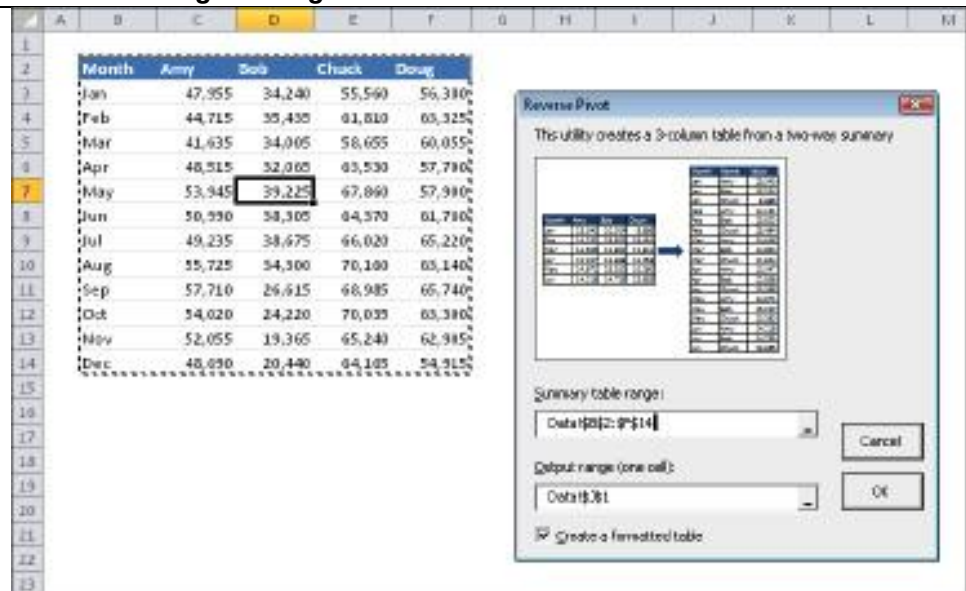


FIGURE 17-9: This dialog box asks the user for the ranges.

When the user clicks the OK button in the UserForm, VBA code validates the ranges and then calls the ReversePivot procedure with this statement:

*Call ReversePivot(SummaryTable, OutputRange, cbCreateTable)*

It passes three arguments:

- **SummaryTable:** A Range object that represents the summary table.
- **OutputRange:** A Range object that represents the upper-left cell of the output range.
- **cbCreateTable:** The Checkbox object on the UserForm.

This procedure will work for any size summary table. The number of data rows in the output table will be equal to  $(r-1) * (c-1)$ , where  $r$  and  $c$  represent the number of rows and columns in the SummaryTable.

The code for the ReversePivot procedure follows:

```

Sub ReversePivot(SummaryTable As Range, _
    OutputRange As Range, CreateTable As Boolean)
    Dim r As Long, c As Long
    Dim OutRow As Long, OutCol As Long
    ' Convert the range
    OutRow = 2
    Application.ScreenUpdating = False
    OutputRange.Range("A1:C3") = Array("Column1", "Column2",
    "Column3")
    For r = 2 To SummaryTable.Rows.Count
    For c = 2 To SummaryTable.Columns.Count
        OutputRange.Cells(OutRow, 1) = SummaryTable.Cells(r, 1)
        OutputRange.Cells(OutRow, 2) = SummaryTable.Cells(1, c)
        OutputRange.Cells(OutRow, 3) = SummaryTable.Cells(r, c)
        OutRow = OutRow + 1
    Next c
    
```

```
Next r
' Make it a table?
If CreateTable Then _
ActiveSheet.ListObjects.Add xlSrcRange, _
OutputRange.CurrentRegion, , xlYes
End Sub
```

The procedure is fairly simple. The code loops through the rows and columns in the input range and then writes the data to the output range. The output range will always have three columns. The OutRow variable keeps track of the current row in the output range. Finally, if the user checked the check box, the output range is converted to a table by using the Add method of the ListObjects collection.

## 9. Appendix

### 9.1. Microsoft Technical Support

Technical support is the common term for assistance provided by a software vendor. In this case, I'm talking about assistance that comes directly from Microsoft. Microsoft's technical support is available in several different forms.

#### Support options

Microsoft's support options are constantly changing. To find out what options are available (both free and fee-based), go to

- <http://support.microsoft.com>

#### Microsoft Knowledge Base

Perhaps your best bet for solving a problem may be the Microsoft Knowledge Base, which is the primary Microsoft product information source. It's an extensive, searchable database that consists of tens of thousands of detailed articles containing technical information, bug lists, fix lists, and more.

You have free and unlimited access to the Knowledge Base via the Internet. To access the Knowledge Base, go to the following URL, enter some search terms, and click Search:

- <http://support.microsoft.com/search>

#### Microsoft Excel home page

The official home page of Excel is at

- [www.microsoft.com/office/excel](http://www.microsoft.com/office/excel)

This site contains a variety of material, such as tips, templates, answers to questions, training materials, and links to companion products.

#### Microsoft Office home page

For information about Office 2010 (including Excel), try this site:

- <http://office.microsoft.com>

You'll find product updates, add-ins, examples, and lots of other useful information.

### 9.2. VBA Statements and Functions Reference

This appendix contains a complete listing of all Visual Basic for Applications (VBA) statements and built-in functions. For details, consult Excel's online help.

**Notes: There are no new VBA statements in Excel 2010.**

**Table B-1: Summary of VBA Statements**

Statement	Action
AppActivate	Activates an application window
Beep	Sounds a tone via the computer's speaker

Call	Transfers control to another procedure
ChDir	Changes the current directory
ChDrive	Changes the current drive
Close	Closes a text file
Const	Declares a constant value
Date	Sets the current system date
Declare	Declares a reference to an external procedure in a Dynamic Link Library (DLL)
DefBool	Sets the default data type to Boolean for variables that begin with specified letters
DefByte	Sets the default data type to Byte for variables that begin with specified letters
DefCur	Sets the default data type to Currency for variables that begin with specified letters
DefDate	Sets the default data type to Date for variables that begin with specified letters
DefDec	Sets the default data type to Decimal for variables that begin with specified letters
DefDbl	Sets the default data type to Double for variables that begin with specified letters
DefInt	Sets the default data type to Integer for variables that begin with specified letters
DefLng	Sets the default data type to Long for variables that begin with specified letters
DefObj	Sets the default data type to Object for variables that begin with specified letters
DefSng	Sets the default data type to Single for variables that begin with specified letters
DefStr	Sets the default data type to String for variables that begin with specified letters
DefVar	Sets the default data type to Variant for variables that begin with specified letters
DeleteSetting	Deletes a section or key setting from an application's entry in the Windows Registry
Dim	Declares variables and (optionally) their data types
Do-Loop	Loops through a set of instructions
End	Used by itself, exits the program; also used to end a block of statements that begin with If, With, Sub, Function, Property, Type, or Select
Enum	Declares a type for enumeration

Erase	Re-initializes an array
Error	Simulates a specific error condition
Event	Declares a user-defined event
Exit Do	Exits a block of Do-Loop code
Exit For	Exits a block of For-Next code
Exit Function	Exits a Function procedure
Exit Property	Exits a property procedure
Exit Sub	Exits a subroutine procedure
FileCopy	Copies a file
For Each-Next	Loops through a set of instructions for each member of a series
For-Next	Loops through a set of instructions a specific number of times
Function	Declares the name and arguments for a Function procedure
Get	Reads data from a text file
GoSub...Return	Branches to and returns from a procedure
GoTo	Branches to a specified statement within a procedure
If-Then-Else	Processes statements conditionally
Implements	Specifies an interface or class that will be implemented in a class module
Input #	Reads data from a sequential text file
Kill	Deletes a file from a disk
Let	Assigns the value of an expression to a variable or property
Line Input #	Reads a line of data from a sequential text file
Load	Loads an object but doesn't show it
Lock...Unlock	Controls access to a text file
Lset	Left-aligns a string within a string variable
Mid	Replaces characters in a string with other characters
MkDir	Creates a new directory
Name	Renames a file or directory
On Error	Gives specific instructions for what to do in the case of an error
On...GoSub	Branches, based on a condition
On...GoTo	Branches, based on a condition
Open	Opens a text file
Option Base	Changes the default lower limit for arrays

Option Compare	Declares the default comparison mode when comparing strings
Option Explicit	Forces declaration of all variables in a module
Option Private	Indicates that an entire module is Private
Print #	Writes data to a sequential file
Private	Declares a local array or variable
Property Get	Declares the name and arguments of a Property Get procedure
Property Let	Declares the name and arguments of a Property Let procedure
Property Set	Declares the name and arguments of a Property Set procedure
Public	Declares a public array or variable
Put	Writes a variable to a text file
RaiseEvent	Fires a user-defined event
Randomize	Initializes the random number generator
ReDim	Changes the dimensions of an array
Rem	Specifies a line of comments (same as an apostrophe ['])
Reset	Closes all open text files
Resume	Resumes execution when an error-handling routine finishes
Rmdir	Removes an empty directory
RSet	Right-aligns a string within a string variable
SaveSetting	Saves or creates an application entry in the Windows Registry
Seek	Sets the position for the next access in a text file
Select Case	Processes statements conditionally
SendKeys	Sends keystrokes to the active window
Set	Assigns an object reference to a variable or property
SetAttr	Changes attribute information for a file
Static	Declares variables at the procedure level so that the variables retain their values as long as the code is running
Stop	Pauses the program
Sub	Declares the name and arguments of a Sub procedure
Time	Sets the system time
Type	Defines a custom data type
Unload	Removes an object from memory
While...Wend	Loops through a set of instructions as long as a certain condition remains true



### 9.3. Invoking Excel functions in VBA instructions

Width #	Sets the output line width of a text file
With	Sets a series of properties for an object
Write #	Writes data to a sequential text file

If a VBA function that's equivalent to one you use in Excel isn't available, you can use Excel's worksheet functions directly in your VBA code. Just precede the function with a reference to the WorksheetFunction object. For example, VBA doesn't have a function to convert radians to degrees. Because Excel has a worksheet function for this procedure, you can use a VBA instruction such as the following:

```
Deg = Application.WorksheetFunction.Degrees(3.14)
```

The WorksheetFunction object was introduced in Excel 97. For compatibility with earlier versions of Excel, you can omit the reference to the WorksheetFunction object and write an instruction such as the following:

```
Deg = Application.Degrees(3.14)
```

**Notes:** There are no new VBA functions in Excel 2010.

**Table B-2: Summary of VBA Functions**

Function	Action
Abs	Returns the absolute value of a number
Array	Returns a variant containing an array
Asc	Converts the first character of a string to its ASCII value
Atn	Returns the arctangent of a number
CallByName	Executes a method, or sets or returns a property of an object
CBool	Converts an expression to a Boolean data type
CByte	Converts an expression to a Byte data type
CCur	Converts an expression to a Currency data type
CDate	Converts an expression to a Date data type
CDbl	Converts an expression to a Double data type
CDec	Converts an expression to a Decimal data type
Choose	Selects and returns a value from a list of arguments
Chr	Converts a character code to a string
CInt	Converts an expression to an Integer data type
CLng	Converts an expression to a Long data type
Cos	Returns the cosine of a number
CreateObject	Creates an Object Linking and Embedding (OLE) Automation object

CSng	Converts an expression to a Single data type
CStr	Converts an expression to a String data type
CurDir	Returns the current path
CVar	Converts an expression to a variant data type
CVDate	Converts an expression to a Date data type (for compatibility, not recommended)
CVErr	Returns a user-defined error value that corresponds to an error number
Date	Returns the current system date
DateAdd	Adds a time interval to a date
DateDiff	Returns the time interval between two dates
DatePart	Returns a specified part of a date
DateSerial	Converts a date to a serial number
DateValue	Converts a string to a date
Day	Returns the day of the month of a date
DDB	Returns the depreciation of an asset
Dir	Returns the name of a file or directory that matches a pattern
DoEvents	Yields execution so the operating system can process other events
Environ	Returns an operating environment string
EOF	Returns True if the end of a text file has been reached
Error	Returns the error message that corresponds to an error number
Exp	Returns the base of natural logarithms (e) raised to a power
FileAttr	Returns the file mode for a text file
FileDateTime	Returns the date and time when a file was last modified
FileLen	Returns the number of bytes in a file
Filter	Returns a subset of a string array, filtered
Fix	Returns the integer portion of a number
Format	Displays an expression in a particular format
FormatCurrency	Returns an expression formatted with the system currency symbol
FormatDateTime	Returns an expression formatted as a date or time
FormatNumber	Returns an expression formatted as a number
FormatPercent	Returns an expression formatted as a percentage

FreeFile	Returns the next available file number when working with text files
FV	Returns the future value of an annuity
GetAllSettings	Returns a list of settings and values from the Windows Registry
GetAttr	Returns a code representing a file attribute
GetObject	Retrieves an OLE Automation object from a file
GetSetting	Returns a specific setting from the application's entry in the Windows Registry
Hex	Converts from decimal to hexadecimal
Hour	Returns the hour of a time
If	Evaluates an expression and returns one of two parts
Input	Returns characters from a sequential text file
InputBox	Displays a box to prompt a user for input
InStr	Returns the position of a string within another string
InStrRev	Returns the position of a string within another string from the end of the string
Int	Returns the integer portion of a number
IPmt	Returns the interest payment for a given period of an annuity
IRR	Returns the internal rate of return for a series of cash flows
IsArray	Returns True if a variable is an array
IsDate	Returns True if a variable is a date
IsEmpty	Returns True if a variable has not been initialized
IsError	Returns True if an expression is an error value
IsMissing	Returns True if an optional argument was not passed to a procedure
IsNull	Returns True if an expression contains a Null value
IsNumeric	Returns True if an expression can be evaluated as a number
IsObject	Returns True if an expression references an OLE Automation object
Join	Combines strings contained in an array
LBound	Returns the smallest subscript for a dimension of an array
LCase	Returns a string converted to lowercase
Left	Returns a specified number of characters from the left of a string
Len	Returns the number of characters in a string

Loc	Returns the current read or write position of a text file
LOF	Returns the number of bytes in an open text file
Log	Returns the natural logarithm of a number
LTrim	Returns a copy of a string with no leading spaces
Mid	Returns a specified number of characters from a string
Minute	Returns the minute of a time
MIRR	Returns the modified internal rate of return for a series of periodic cash flows
Month	Returns the month of a date as a number
MonthName	Returns the month of a date as a string
MsgBox	Displays a modal message box
Now	Returns the current system date and time
NPer	Returns the number of periods for an annuity
NPV	Returns the net present value of an investment
Oct	Converts from decimal to octal
Partition	Returns a string representing a range in which a value falls
Pmt	Returns a payment amount for an annuity
Ppmt	Returns the principal payment amount for an annuity
PV	Returns the present value of an annuity
QBColor	Returns a red/green/blue (RGB) color code
Rate	Returns the interest rate per period for an annuity
Replace	Returns a string in which a substring is replaced with another string
RGB	Returns a number representing an RGB color value
Right	Returns a specified number of characters from the right of a string
Rnd	Returns a random number between 0 and 1
Round	Returns a rounded number
RTrim	Returns a copy of a string with no trailing spaces
Second	Returns the seconds portion of a specified time
Seek	Returns the current position in a text file
Sgn	Returns an integer that indicates the sign of a number
Shell	Runs an executable program
Sin	Returns the sine of a number

SLN	Returns the straight-line depreciation for an asset for a period
Space	Returns a string with a specified number of spaces
Spc	Positions output when printing to a file
Split	Returns a one-dimensional array containing a number of substrings
Sqr	Returns the square root of a number
Str	Returns a string representation of a number
StrComp	Returns a value indicating the result of a string comparison
StrConv	Returns a converted string
String	Returns a repeating character or string
StrReverse	Returns a string, reversed
Switch	Evaluates a list of Boolean expressions and returns a value associated with the first True expression
SYD	Returns the sum-of-years' digits depreciation of an asset for a period
Tab	Positions output when printing to a file
Tan	Returns the tangent of a number
Time	Returns the current system time
Timer	Returns the number of seconds since midnight
TimeSerial	Returns the time for a specified hour, minute, and second
TimeValue	Converts a string to a time serial number
Trim	Returns a string without leading spaces and/or trailing spaces
TypeName	Returns a string that describes the data type of a variable
UBound	Returns the largest available subscript for a dimension of an array
UCase	Converts a string to uppercase
Val	Returns the number formed from any initial numeric characters of a string
VarType	Returns a value indicating the subtype of a variable
Weekday	Returns a number indicating a day of the week
WeekdayName	Returns a string indicating a day of the week
Year	Returns the year of a date

#### 9.4. VBA Error Codes

This appendix contains a complete listing of the error codes for all trappable errors in Visual Basic for Applications (VBA). This information is useful for error trapping. For complete details, consult Excel's Help system.

Error Code	Message
3	Return without GoSub.
5	Invalid procedure call or argument.
6	Overflow (for example, value too large for an integer).
7	Out of memory. This error rarely refers to the amount of physical memory installed on your system. Rather, it usually refers to a fixed-size area of memory used by Excel or Windows (for example, the area used for graphics or custom formats).
9	Subscript out of range. You will also get this error message if a named item is not found in a collection of objects. For example, if your code refers to Sheets("Sheet2"), and Sheet2 does not exist.
10	This array is fixed or temporarily locked.
11	Division by zero.
13	Type mismatch.
14	Out of string space.
16	Expression too complex.
17	Can't perform requested operation.
18	User interrupt occurred. This error occurs if the user interrupts a macro by pressing the Cancel key.
20	Resume without error. This error probably indicates that you forgot the Exit Sub statement before your error handler code.
28	Out of stack space.
35	Sub or Function not defined.
47	Too many Dynamic Link Library (DLL) application clients.
48	Error in loading DLL.
49	Bad DLL calling convention.
51	Internal error.
52	Bad filename or number.
53	File not found.
54	Bad file mode.
55	File already open.
57	Device Input/Output (I/O) error.
58	File already exists.
59	Bad record length.
61	Disk full.
62	Input past end of file.

63	Bad record number.
67	Too many files.
68	Device unavailable.
70	Permission denied.
71	Disk not ready.
74	Can't rename with different drive.
75	Path/File access error.
76	Path not found.
91	Object variable or With block variable not set. This error occurs if you don't use Set at the beginning of a statement that creates an object variable. Or, it occurs if you refer to a worksheet object (such as ActiveCell) when a chart sheet is active.
92	For loop not initialized.
93	Invalid pattern string.
94	Invalid use of Null.
96	Unable to sink events of object because the object is already firing events to the maximum number of event receivers that it supports.
97	Cannot call friend function on object that is not an instance of defining class.
98	A property or method call can't include a reference to a private object, either as an argument or as a return value.
321	Invalid file format.
322	Can't create necessary temporary file.
325	Invalid format in resource file.
380	Invalid property value.
381	Invalid property array index.
382	Set not supported at runtime.
383	Set not supported (read-only property).
385	Need property array index.
387	Set not permitted.
393	Get not supported at runtime.
394	Get not supported (write-only property).
422	Property not found.
423	Property or method not found.
424	Object required. This error occurs if text preceding a dot is not recognized as an object.

429	ActiveX component can't create object (might be a registration problem with a library that you've referenced).
430	Class doesn't support Automation or doesn't support expected interface.
432	Filename or class name not found during Automation operation.
438	Object doesn't support this property or method.
440	Automation error.
442	Connection to type library or object library for remote process has been lost.
443	Automation object doesn't have a default value.
445	Object doesn't support this action.
446	Object doesn't support named arguments.
447	Object doesn't support current locale setting.
448	Named argument not found.
449	Argument not optional.
450	Wrong number of arguments or invalid property assignment.
451	Property Let procedure not defined, and Property Get procedure did not return an object.
452	Invalid ordinal.
453	Specified DLL function not found.
454	Code resource not found.
455	Code resource lock error.
457	Key is already associated with an element of this collection.
458	Variable uses an Automation type not supported in Visual Basic.
459	Object or class doesn't support the set of events.
460	Invalid Clipboard format.
461	Method or data member not found.
462	Remote server machine doesn't exist or is unavailable.
463	Class not registered on local machine.
481	Invalid picture.
482	Printer error.
735	Can't save file to TEMP.
744	Search text not found.
746	Replacements too long.



1004	Application-defined or object-defined error. This is a very common catch-all error message. This error occurs when an error doesn't correspond to an error defined by VBA. In other words, the error is defined by Excel (or some other object) and is propagated back to VBA.
------	--



