Programmer to Programmer™

Excel® 2007 VBA

Programmer's Reference

John Green, Stephen Bullen, Rob Bovey, Michael Alexander

wrox™

Updates, source code, and Wrox technical support at **www.wrox.com**

Chapter 1

Primer in Excel VBA

This chapter is intended for those who are not familiar with Excel and the Excel macro recorder, or who are inexperienced with programming using the Visual Basic language. If you are already comfortable with navigating around the features provided by Excel, have used the macro recorder, and have a working knowledge of Visual Basic and the Visual Basic Editor, you might want to skip straight to Chapter 2.

If this is not the case, this chapter has been designed to provide you with the information you need to be able to move on comfortably to the more advanced features presented in the following chapters. Specifically, this chapter covers the following topics:

- The Excel macro recorder
- User-defined functions
- The Excel object model
- VBA programming concepts

Excel VBA is a programming application that allows you to use Visual Basic code to run the many features of the Excel package, thereby allowing you to customize your Excel applications. Units of VBA code are often referred to as *macros.* More formal terminology is covered in this chapter, but you will continue to see the term *macro* as a general way to refer to any VBA code.

In your day-to-day use of Excel, if you carry out the same sequence of commands repetitively, you can save a lot of time and effort by automating those steps using macros. If you are setting up an application for other users who don't know much about Excel, you can use macros to create buttons and dialog boxes to guide them through your application as well as automate the processes involved.

If you are able to perform an operation manually, you can use the *macro recorder* to capture that operation. This is a very quick and easy process and requires no prior knowledge of the VBA language. Many Excel users record and run macros and feel no need to learn about VBA.

However, the recorded results might not be very flexible, in that the macro can only be used to carry out one particular task on one particular range of cells. In addition, the recorded macro is likely to run much more slowly than code written by someone with knowledge of VBA. To set up interactive macros that can adapt to change and also run quickly, and to take advantage of more advanced features of Excel such as customized dialog boxes, you need to learn about VBA.

> **Don't get the impression that we are dismissing the macro recorder. The macro recorder is one of the most valuable tools available to VBA programmers. It is the fastest way to generate working VBA code, but you must be prepared to apply your own knowledge of VBA to edit the recorded macro to obtain flexible and efficient code. A recurring theme in this book is recording an Excel macro and then showing how to adapt the recorded code.**

In this chapter, you learn how to use the macro recorder and you see all the ways Excel provides to run your macros. You see how to use the *Visual Basic Editor* to examine and change your macros, thus going beyond the recorder and tapping into the power of the VBA language and the *Excel object model.*

You can also use VBA to create your own worksheet functions. Excel comes with hundreds of built-in functions, such as `SUM` and `IF`, which you can use in cell formulas. However, if you have a complex calculation that you use frequently and that is not included in the set of standard Excel functions—such as a tax calculation or a specialized scientific formula—you can write your own *user-defined function.*

Using the Macro Recorder

Excel's macro recorder operates very much like the recorder that stores the greeting on your telephone answering machine. To record a greeting, you first prepare yourself by rehearsing the greeting to ensure that it says what you want. Then you switch on the recorder and deliver the greeting. When you have finished, you switch off the recorder. You now have a recording that automatically plays when you leave a call unanswered.

Recording an Excel macro is very similar. You first rehearse the steps involved and decide at what points you want to start and stop the recording process. You prepare your spreadsheet, switch on the Excel recorder, carry out your Excel operations, and switch off the recorder. You now have an automated procedure that you and others can reproduce at the press of a button.

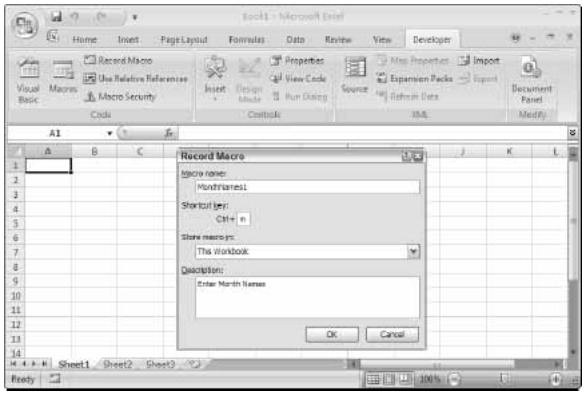
Recording Macros

Say you want a macro that types six month names as three-letter abbreviations, Jan to Jun, across the top of your worksheet, starting in cell B1. I know this is rather a silly macro because you could do this easily with an AutoFill operation, but this example will serve to show you some important general concepts:

- First, think about how you are going to carry out this operation. In this case, it is easy—you will just type the data across the worksheet. Remember, a more complex macro might need more rehearsals before you are ready to record it.
- Next, think about when you want to start recording. In this case, you should include the selection of cell B1 in the recording, because you want to always have Jan in B1. If you don't select B1 at the start, you will record typing Jan into the active cell, which could be anywhere when you play back the macro.
- Next, think about when you want to stop recording. You might first want to include some formatting such as making the cells bold and italic, so you should include that in the recording. Where do you want the active cell to be after the macro runs? Do you want it to be in the same cell as Jun, or would you rather have the active cell in column A or column B, ready for your next input? Assume that you want the active cell to be A2, at the completion of the macro, so you will select A2 before turning off the recorder.
- Now you can set up your screen, ready to record.

In this case, start with an empty worksheet with cell A1 selected. If you can't see the Developer tab above the Ribbon, you will need to click the round Microsoft Office button that you can see in the top-left corner of the Excel screen shown in Figure 1-1. Click Excel Options at the bottom of the dialog box and select Personalize. Select the checkbox for Show Developer tab in the Ribbon and click OK. Now you can select the Developer section of the Ribbon and click Record Macro to display the Record Macro dialog box, shown in Figure 1-1.

Figure 1-1



In the Macro name: box, replace the default entry, such as Macro1, with the name you want for your macro. The name should start with a letter and contain only letters, numbers, and the underscore character, with a maximum length of 255 characters. The macro name must not contain special characters such as exclamation points (!) or question marks (?), nor should it contain blank spaces. It is also best to use a short but descriptive name that you will recognize later. You can use the underscore character to separate words, but it is easy to just use capitalization to distinguish words.

Call the macro `MonthNamesl`, because you will create another version later.

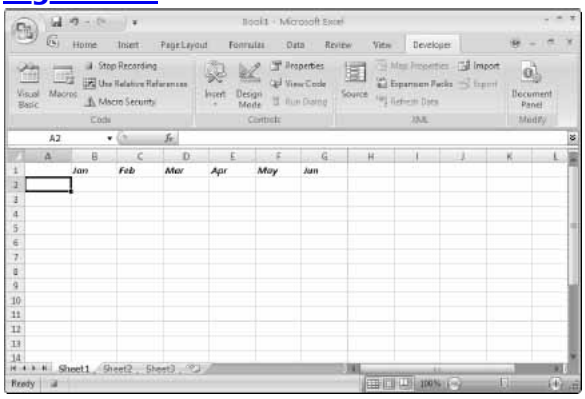
In the Shortcut key: box, you can type in a single letter. This key can be pressed later, while holding down the Ctrl key, to run the macro. Use a lowercase m. Alternatively, you can use an uppercase M. In this case, when you later want to run the macro, you need to use the keystroke combination Ctrl+Shift+M. It is not mandatory to provide a shortcut key; you can run a macro in a number of other ways, as you will see.

In the Description: box, you can add text that will be added as comments to the macro. These lines will appear at the top of your macro code. They have no significance to VBA, but provide you and others with information about the macro.

All Excel macros are stored in workbooks. You are given a choice regarding where the recorded macro will be stored. The Store macro in: combo box lists three possibilities. If you choose New Workbook, the recorder will open a new empty workbook for the macro. Personal Macro Workbook refers to a special hidden workbook, which is discussed in a moment. Choose This Workbook to store the macro in the currently active workbook.

When you have filled in the Record Macro dialog box, click the OK button. You will see a new Stop Recording button appear on the left side of the status bar at the bottom of the screen, as shown in Figure 1-2. You will also notice that the Start Recording button in the Ribbon has been replaced by a new Stop Recording button.

Figure 1-2



You should now click cell B1, type in Jan, and fill in the rest of the cells as shown in Figure 1-2. Then select B1:G1 and click the Bold and Italic buttons on the Home tab of the Ribbon. Click the A2 cell and then stop the recorder. You can stop the recorder by clicking the Stop Recording button on the Ribbon or by clicking the Stop Recording button on the status bar.

> **It is important to remember to stop the recorder. If you leave the recorder on and try to run the recorded macro, you can go into a loop where the macro runs itself over and over again. If this does happen to you, or any other error occurs while testing your macros, hold down the Ctrl key and press the Break key to interrupt the macro. You can then end the macro or go into debug mode to trace errors. You can also interrupt a macro with the Esc key, but it is not as effective as Ctrl+Break for a macro that is pausing for input.**

You could now save the workbook, but before you do so, you should determine the file type you need and consider the security issues covered in the next section.

You can't save the workbook as the default Excel Workbook (*.xlsx) type. This file format does not allow macros to be included. You can save the workbook as an Excel Macro-Enabled Workbook (*.xlsm) type, which is in XLM format, or you can save it as an Excel Binary Workbook (*.xlsb) type, which is in a binary format. Neither of these file types is compatible with previous versions of Excel. Another alternative is to save the workbook as an Excel 97-2003 Workbook (*.xls) type, which produces a workbook compatible with Excel versions from Excel 97 through Excel 2003.

Macro Security

To develop macros with minimum interruption, work with Office 2007's security restrictions. Without getting into the complications of digitally signing your workbooks, you have a couple of simple options. Select the Developer tab on the Ribbon and click the Macro Security button. You will see the Trust Center dialog box, where you can select Macro Settings. Here you can enable all macros. This is not recommended because it leaves you wide open to macro viruses.

A better alternative is to nominate a specific directory as a trusted location. Click Trusted Locations to the left of the Trust Center dialog box. You probably already have a number of trusted locations, including your XLSTART directory and templates directories. Use the Add new location button to specify a suitable directory for storing your workbooks.

You should now save the workbook containing the newly recorded macro into the trusted location. Click the Microsoft Office button and select Save As. In the Save as type drop-down, select the .xlsm type and save the workbook in the trusted location as `Recorder.xlsm`.

> **If you can't see the file extensions, such as `.xlsm`, in the Save As dialog box, you should open Windows Explorer, click the Tools menu, and choose Folder Options. In the View tab, remove the check against Hide extensions for known file types.**

The Personal Macro Workbook

If you choose to store your recorded macro in the Personal Macro Workbook, the macro is added to a special file called `Personal.xlsb`, which is a hidden file that is saved in your Excel Startup directory when you close Excel. This means that `Personal.xlsb` is automatically loaded when you launch Excel and, therefore, its macros are always available for any other workbook to use.

If `Personal.xlsb` does not already exist, the recorder will create it for you. You can use the Unhide button on the View tab of the Ribbon to see this workbook in the Excel window, but

it is seldom necessary or desirable to do this because you can examine and modify the `Personal.xlsb` macros in the Visual Basic Editor window.

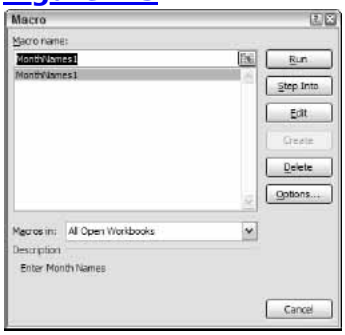
An exception where you might want to make `Personal.xlsb` visible is if you need to store data in its worksheets. You can hide it again, after adding the data, with the Hide button on the View tab of the Ribbon. If you are creating a general-purpose utility macro, which you want to be able to use with any workbook, store it in `Personal.xlsb`. If the macro relates to just the application in the current workbook, store the macro with the application.

Running Macros

To run the macro, either use another worksheet in the `Recorder.xlsm` workbook or open a new empty workbook, leaving `Recorder.xlsm` open in memory. You can only run macros that are in open workbooks, but they can be run from within any other open workbook.

You can run the macro by pressing Ctrl+M, the shortcut you assigned at the start of the recording process. You can also run the macro by clicking the Macros button in the View tab of the Ribbon or by clicking the Macros button in the Developer tab of the Ribbon. Both buttons open the dialog box shown in Figure 1-3. You can run the macro by double-clicking the macro name, or by selecting the macro name and clicking Run.

Figure 1-3

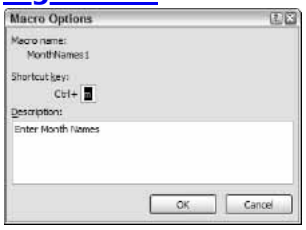


The same dialog box can be opened by pressing Alt+F8.

Shortcut Keys

You can change the shortcut key assigned to a macro using the Macro dialog box shown in Figure 1-3. Select the macro name and click Options. This opens the dialog box shown in Figure 1-4.

Figure 1-4



It is possible to assign the same shortcut key to more than one macro in the same workbook using this dialog box (although the dialog box that appears when you start the macro recorder will not let you assign a shortcut that is already in use).

It is also quite likely that two different workbooks could contain macros with the same shortcut key assigned. If this happens, which macro runs when you use the shortcut? The macro that comes first alphabetically.

Shortcuts are appropriate for macros that you use frequently, especially if you prefer to keep your hands on the keyboard. It is worth memorizing the shortcuts so you won't forget them if you use them regularly. Shortcuts are *not* appropriate for macros that are run infrequently or are intended to make life easier for less experienced users of your application. It is better to assign meaningful names to those macros and run them from the Macro dialog box. Alternatively, they can be run from buttons that you add to the worksheet. You learn how to do this shortly.

Absolute and Relative Recording

When you run `MonthNamesl`, the macro returns to the same cells you selected while typing in the month names. It doesn't matter which cell is active when you start; if the macro contains the command to select cell B1, that is what it selects. The macro selects B1 because you recorded in absolute record mode. The alternative, relative record mode, remembers the position of the active cell relative to its previous position. If you have cell A10 selected, turn on the recorder, and go on to select B10, the recorder notes that you moved one cell to the right, rather than noting that you selected cell B10.

Record a second macro called `MontlhNames2`. There will be three differences in this macro compared with the previous one:

- Click the Use Relative References button on the Developer tab of the Ribbon. You can do this before you start recording or while you are recording.
- Do not select the Jan cell before typing. You want your recorded macro to type Jan into the active cell when you run the macro.
- Finish by selecting the cell under Jan, rather than A2, just before turning off the recorder.

Start with an empty worksheet and select the B1 cell. Turn on the macro recorder and specify the macro name as `MonthNames2`. Enter the shortcut as uppercase M—the recorder won't let you use lowercase m again. Click OK and select the Use Relative References button on the Developer tab of the Ribbon.

Type Jan and the other month names, as you did when recording `MonthNamesl`. Select cells B1:G1 and click the Bold and Italic buttons on the Home tab of the Ribbon.

> **Make sure you select B1:G1 from left to right, so that B1 is the active cell. There is a small kink in the recording process that can cause errors in the recorded macro if you select cells from right to left or from bottom to top. Always select from the top-left corner when recording relatively. This has been a problem with all versions of Excel VBA.**

Finally, select cell B2, the cell under Jan, and turn off the recorder.

Before running `MonthNames2`, select a starting cell, such as A10. You will find that the macro now types the month names across row 10, starting in column A, and finishes by selecting the cell under the starting cell.

Before you record a macro that selects cells, you need to think about whether to use absolute or relative reference recording. If you are selecting input cells for data entry, or for a print area, you will probably want to record with absolute references. If you want to be able to run your macro in different areas of your worksheet, you will probably want to record with relative references.

If you are trying to reproduce the effect of the Ctrl+arrow keys to select the last cell in a column or row of data, you should record with relative references. You can even switch between relative and absolute reference recording in the middle of a macro, if you want. You might want to select the top of a column with an absolute reference, switch to relative references, and use Ctrl+down arrow to get to the bottom of the column and an extra down arrow to go to the first empty cell.

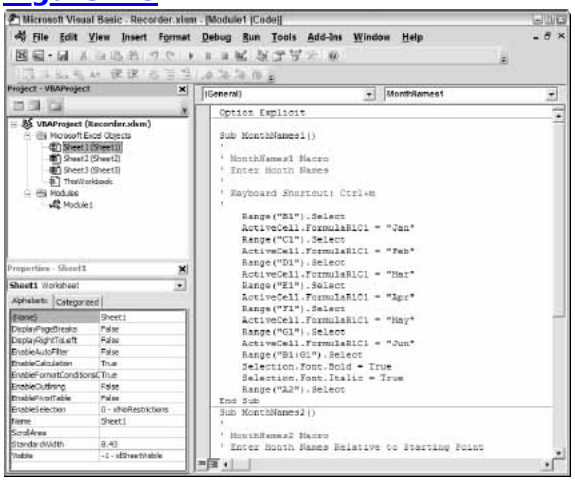
Excel 2000 was the first version of Excel to let you successfully record selecting a block of cells of variable height and width using the Ctrl key. If you start at the top left-hand corner of a block of data, you can hold down the Shift and Ctrl keys and press the down arrow and then the right arrow to select the whole block (as long as there are no gaps in the data). If you record these operations with relative referencing, you can use the macro to select a block of different dimensions. Previous versions of Excel recorded an absolute selection of the original block size, regardless of recording mode.

The Visual Basic Editor

It is now time to see what has been going on behind the scenes. If you want to understand macros, be able to modify your macros, and tap into the full power of VBA, you need to know how to use the Visual Basic Editor (VBE). The VBE runs in its own window, separate from the Excel window. You can activate it in many ways.

First, you can activate it by clicking the Visual Basic button on the Developer tab of the Ribbon. You can also activate it by holding down the Alt key and pressing the F11 key. Alt+F11 acts as a toggle, taking you between the Excel window and the VBE window. If you want to edit a specific macro, you can use the Macros button on the Developer tab of the Ribbon or the Play Macro button on the left of the status bar to open the Macro dialog box, select the macro, and click the Edit button. The VBE window will look something like Figure 1-5.

Figure 1-5



It is quite possible that you will see nothing but the menu bar when you switch to the VBE window. If you can't see the toolbars, use View ➪ Toolbars and click the Standard toolbar. Use View ➪ Project Explorer and View ➪ Properties Window to show the windows on the left. If you can't see the code module on the right, double-click the icon for Module1 in the Project Explorer window.

Code Modules

All macros reside in code modules like the one on the right of the VBE window in Figure 1-5. There are two types of code modules—standard modules and class modules. The one you see on the right is a standard module. You can use class modules to create your own objects. You won't need to know much about class modules until you are working at a very advanced level. See Chapter 15 for more details on how to use class modules.

Some class modules have already been set up for you. They are associated with each worksheet in your workbook, and there is one for the entire workbook. You can see them in the Project Explorer window, in the folder called Microsoft Excel Objects. You will find out more about them later in this chapter.

You can add as many code modules to your workbook as you like. The macro recorder has inserted the one named Module1. Each module can contain many macros. For a small application, you would probably keep all your macros in one module. For larger projects, you can organize your code better by filing unrelated macros in separate modules.

Procedures

In VBA, macros are referred to as procedures. There are two types of procedures—sub procedures and function procedures. You will find out about function procedures in the next section. The macro recorder can only produce sub procedures. You can see the `MonthNamesl` sub procedure set up by the recorder in Figure 1-5.

Sub procedures start with the keyword `Sub`, followed by the name of the procedure and opening and closing parentheses. The end of a sub procedure is marked by the keywords `End Sub`. Although it is not mandatory, the code within the sub procedure is normally indented to make it stand out from the start and end of the procedure, so that the whole procedure is easier to read. Further indentation is normally used to distinguish sections of code such as `If` tests and looping structures. For example:

```
If ActiveCell.Value = 10 Then
   ActiveCell.Font.Bold = True
End If
```

Any lines starting with a single quote are comment lines, which are ignored by VBA. They are added to provide documentation, which is a very important component of good programming practice. You can also add comments to the right of lines of code. For example:

```
Range("B1").Select   'Select the B1 cell
```

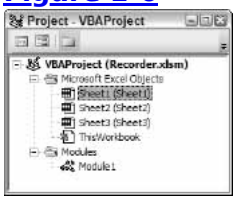
At this stage, the code may not make perfect sense, but you should be able to make out roughly what is going on. If you look at the code in `MonthNamesl`, you will see that cells are being selected and then the month names are assigned to the active cell formula. You can edit some parts of the code, so if you had spelled a month name incorrectly, you could fix it; or you could identify and remove the line that sets the font to bold; or you can select and delete an entire macro.

Notice the differences between `MonthNamesl` and `MonthNames2`. `MonthNamesl` selects specific cells such as B1 and C1. `MonthNames2` uses `Offset` to select a cell that is zero rows down and one column to the right from the active cell. Already, you are starting to get a feel for the VBA language.

The Project Explorer

The Project Explorer is an essential navigation tool. In VBA, each workbook contains a project. The Project Explorer displays all the open projects and the component parts of those projects, as you can see in Figure 1-6.

Figure 1-6



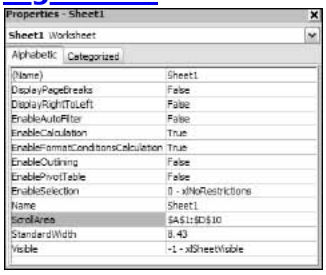
You can use the Project Explorer to locate and activate the code modules in your project. You can doubleclick a module icon to open and activate that module. You can also insert and remove code modules in the Project Explorer. Right-click anywhere in the Project Explorer window, and from the context menu select Insert to add a new standard module, class module, or UserForm.

To remove Module1, right-click it and choose Remove Module1. Note that you can't do this with the modules associated with workbook or worksheet objects. You can also export the code in a module to a separate text file, or import code from a text file.

The Properties Window

The Properties window shows you the properties that can be changed at design time for the currently active object in the Project Explorer window. For example, if you click Sheet1 in the Project Explorer, the Sheet1 properties are displayed in the Properties window, as shown in Figure 1-7. The `ScrollArea` property has been set to `A1:D10`, to restrict users to that area of the worksheet.

Figure 1-7



You can get to the help screen associated with any property very easily. Just select the property, such as the `ScrollArea` property, which is selected in Figure 1-7, and press F1.

Other Ways to Run Macros

You have seen how to run macros with shortcuts and how to run them from the Ribbon and status bar macro buttons. Neither method is particularly friendly. You need to be very familiar with your macros to be comfortable with these techniques. You can make your macros much more accessible by attaching them to buttons.

If the macro is worksheet-specific, and will only be used in a particular part of the worksheet, then it is suitable to use a button that has been embedded in the worksheet at the appropriate location. If you want to be able to use a macro in any worksheet or workbook and in any location in a worksheet, it is appropriate to attach the macro to a button on the Quick Access Toolbar.

There are many other objects that you can attach macros to, including combo boxes, list boxes, scrollbars, checkboxes, and option buttons. These are all referred to as controls. (See Chapter 11 for more information on controls.) You can also attach macros to graphic objects in the worksheet, such as shapes created with the Shapes button on the Insert tab of the Ribbon.

Worksheet Buttons

Excel 2007 has two different sets of controls that can be embedded in worksheets. One set has been inherited from the Forms toolbar in previous versions, and the other has been inherited from the Control ToolBox toolbar in previous versions. The Forms toolbar appeared in Excel 5 and 95. The Forms controls can be embedded in a worksheet and are also used with Excel 5 and 95 dialog sheets to create dialog boxes. Excel 97 introduced the newer ActiveX controls on the Control ToolBox toolbar. You can embed ActiveX controls in a worksheet or use them on UserForms, in the VBE, to create dialog boxes.

To create controls in Excel 2007, select the Developer tab on the Ribbon. In the Controls group, click the Insert button to open the window shown in Figure 1-8.

Figure 1-8

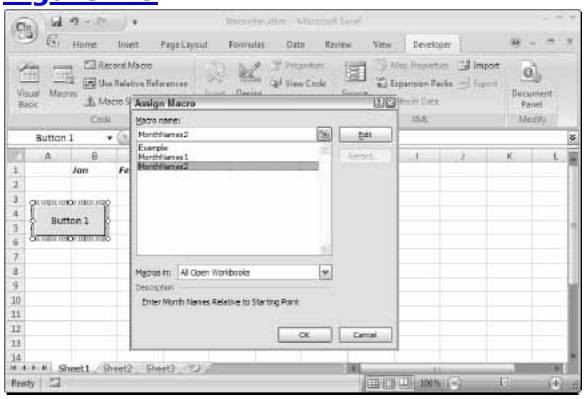
For compatibility with the older versions of Excel, both sets of controls and techniques for creating dialog boxes are supported in Excel 97 and higher. If you have no need to maintain backward compatibility with Excel 5 and 95, you can use just the ActiveX controls.

Forms Controls

A good reason for using the Forms controls is that they are simpler to use than the ActiveX controls, because they do not have all the features of ActiveX controls. For example, Forms controls can only respond to a single, predefined event, which is usually the mouse-click event. ActiveX controls can respond to many events, such as a mouse click, a double-click, or pressing a key on the keyboard. If you have no need of such features, you might prefer the simplicity of Forms controls. To create a Forms button in a worksheet, click the top-left button in the Controls dialog box, opened from the Insert button on the Developer tab of the Ribbon.

You can now draw the button in your worksheet by clicking where you want a corner of the button to appear and dragging to where you want the diagonally opposite corner to appear. The Assign Macro dialog box will appear as shown in Figure 1-9, and you can select the macro to attach to the button.

Figure 1-9



Click OK to complete the assignment. You can then edit the text on the button to give a more meaningful indication of its function. After you click a worksheet cell, you can click the button to run the attached macro. If you need to edit the button and it is not already selected, right-click it to select the control and display a context menu. If you don't want the context menu, hold down Ctrl and left-click or right-click the button to select it. (Don't drag the mouse while you hold down Ctrl, or you will create a copy of the button.)

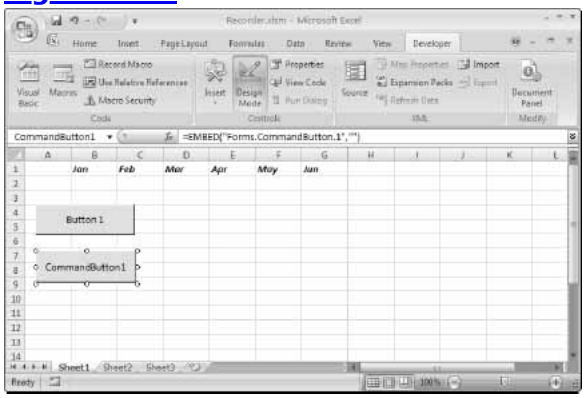
If you want to align the button with the worksheet gridlines, hold down Alt as you draw it with the mouse. If you have already drawn the button, select it and hold down Alt as you drag any of the white boxes that appear on the corners and edges of the button. The edge or corner you drag will snap to the nearest gridline.

ActiveX Controls

To create an ActiveX command button control, click the top-left button in the ActiveX Controls section of the Controls dialog box, opened from the Insert button on the Developer tab of the Ribbon. When you draw your button in the worksheet, you enter into design mode. When you are in design mode, you can select a control with a left-click and edit it. You must turn off design mode if you want the new control to respond to events. You can do

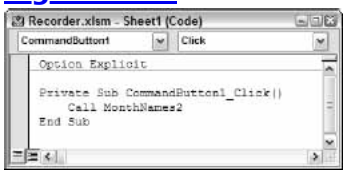
this by clicking the Design Mode button on the Developer tab of the Ribbon so it is no longer highlighted. Figure 1-10 shows the Design Mode button as it appears when design mode is active, after the insertion of the ActiveX control.

Figure 1-10



You are not prompted to assign a macro to the ActiveX command button, but you do need to write a click-event procedure for the button. An event procedure is a sub procedure that is executed when, for example, you click a button. To do this, make sure you are still in design mode and double-click the command button to open the VBE window and display the code module behind the worksheet. The Sub and `End Sub` statement lines for your code will have been inserted in the module, and you can add in the code necessary to run the `MonthNames2` macro, as shown in Figure 1-11.

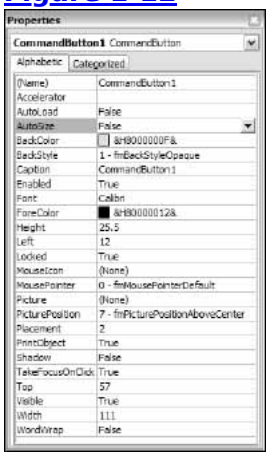
Figure 1-11



To run this code, switch back to the worksheet, turn off design mode, and click the command button.

If you want to make changes to the command button, you need to return to design mode by clicking the Design Mode button. You can then select the command button and change its size and position on the worksheet. You can also display its properties by right-clicking it and choosing Properties to display the window shown in Figure 1-12.

Figure 1-12

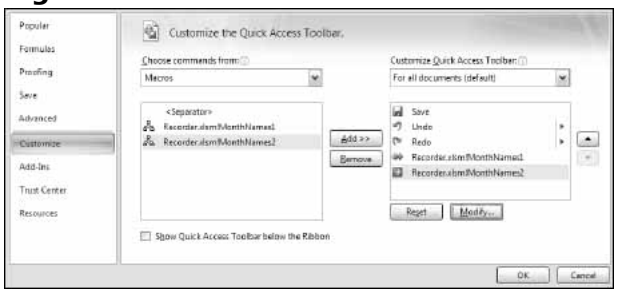


To change the text on the command button, change the `Caption` property. You can also set the font for the caption and the foreground and background colors. If you want the button to work satisfactorily in Excel 97, it is a good idea to change the `TakeFocusOnClick` property from its default value of `True` to `False`. If the button takes the focus when you click it, Excel 97 does not allow you to assign values to some properties, such as the `NumberFormat` property of the `Range` object.

Quick Access Toolbar

In versions of Excel prior to Excel 2007, you can attach macros to toolbar buttons. Because toolbars and menus have been replaced by the Ribbon in Excel 2007, this ability no longer exists, with the exception of the Quick Access Toolbar. The Quick Access Toolbar sits either above or below the Ribbon, and you can add any button from the Ribbon to it to give you direct access to the button. When you right-click a Ribbon button, you can choose Add to Quick Access Toolbar from the pop-up menu. The same pop-up menu offers a second choice, which is Customize Quick Access Toolbar. This choice opens the dialog box shown in Figure 1-13.

Figure 1-13



Select Macros from the Choose commands from: drop-down menu. You can now assign macros from open workbooks to the Quick Access Toolbar by selecting them and clicking the Add button. The icon associated with the macro can be changed by clicking the Modify button, which provides a selection of icons and a text box where you can enter a quick tip for the button.

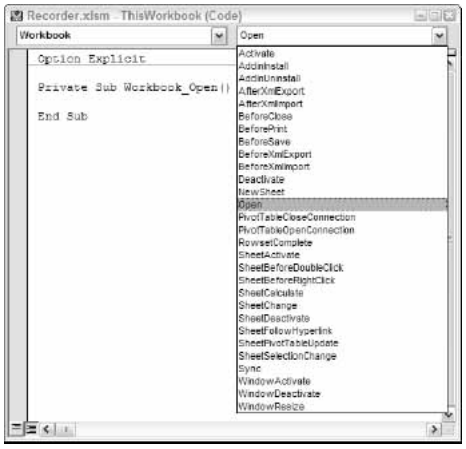
Event Procedures

Event procedures are special macro procedures that respond to the events that occur in Excel. Events include user actions, such as clicking the mouse on a button, and system actions, such as the recalculation of a worksheet. Versions of Excel since Excel 97 expose a wide range of events for which you can write code.

The click-event procedure for the ActiveX command button that ran the `MonthNames2` macro, which you have already seen, is a good example. You entered the code for this event procedure in the code module behind the worksheet where the command button was embedded. All event procedures are contained in the class modules behind the workbook, worksheets, charts, and UserForms.

You can see the events that are available by activating a module, such as the ThisWorkbook module, choosing an object, such as Workbook, from the left drop-down list at the top of the module, and then activating the right drop-down, as shown in Figure 1-14.

Figure 1-14



The `Workbook_Open()` event can be used to initialize the workbook when it is opened. The code could be as simple as activating a particular worksheet and selecting a range for data input. The code could also be more sophisticated and construct new buttons in the Ribbon.

> **For compatibility with Excel 5 and 95, you can still create a sub procedure called `Auto_Open()`, in a standard module, that runs when the workbook is opened. If you also have a `Workbook_Open()` event procedure, the event procedure runs first.**

As you can see, there are many events to choose from. Some events, such as the `BeforeSave` and `BeforeClose` events, allow cancellation of the event. The following event procedure stops the workbook from being closed until cell A1 in Sheet1 contains the value `True`:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
  If ThisWorkbook.Sheets("Sheet1").Range("A1").Value <> True Then
     Cancel = True
  End If
End Sub
```

This code even prevents the closure of the Excel window.

User-Defined Functions

Excel has hundreds of built-in worksheet functions that you can use in cell formulas. You can select an empty worksheet cell, select the Formulas tab of the Ribbon, and click one of the buttons in the Function Library chunk to see a list of functions. Among the most frequently used functions are `SUM`, `IF`, and `VLOOKUP`. If the function you need is not already in Excel, you can write your own user-defined function (or UDF) using VBA.

UDFs can reduce the complexity of a worksheet. It is possible to reduce a calculation that requires many cells of intermediate results down to a single function call in one cell. UDFs can also increase productivity when many users have to repeatedly use the same calculation procedures. You can set up a library of functions tailored to your organization.

Creating a UDF

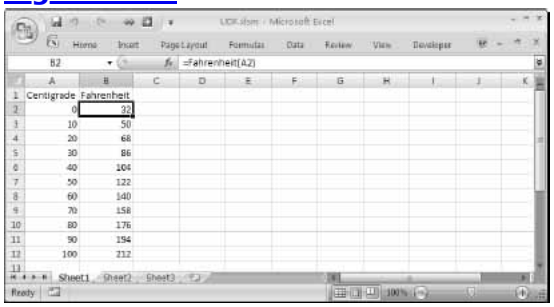
Unlike manual operations, UDFs cannot be recorded—you have to write them from scratch using a standard module in the VBE. If necessary, you can insert a standard module by right-clicking in the Project Explorer window and choosing Insert ➪ Module. A simple example of a UDF is shown here:

```
Function Fahrenheit(Centigrade)
   Fahrenheit = Centigrade * 9 / 5 + 32
End Function
```

Here, a function called `Fahrenheit()` is created that converts degrees Centigrade to degrees Fahrenheit. In the worksheet, you could have column A containing degrees Centigrade and

column B using the `Fahrenheit()` function to calculate the corresponding temperature in degrees Fahrenheit. You can see the formula in cell B2 by looking at the Formula bar in Figure 1-15.

Figure 1-15



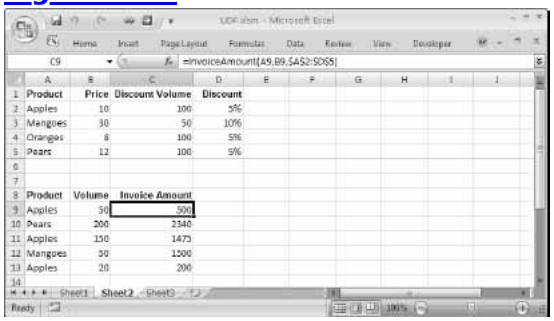
The formula has been copied into cells B3:B12.

The key difference between a sub procedure and a function procedure is that a function procedure returns a value. `Fahrenheit()` calculates a numeric value, which is returned to the worksheet cell where `Fahrenheit()` is used. A function procedure indicates the value to be returned by setting its own name equal to the return value.

Function procedures normally have one or more input parameters. `Fahrenheit()` has one input parameter called `Centigrade`, which is used to calculate the return value. When you enter the formula, `Fahrenheit(A2)`, the value in cell A2 is passed to `Fahrenheit()` through `Centigrade`. In the case where the value of `Centigrade` is 0, `Fahrenheit()` sets its own name equal to the calculated result, which is 32. The result is passed back to cell B2, as shown in Figure 1-15. The same process occurs in each cell that contains a reference to `Fahrenheit()`.

A different example that shows how you can reduce the complexity of spreadsheet formulas for users is shown in Figure 1-16. The lookup table in cells A1:D5 gives the price of each product, the discount sales volume (above which a discount will be applied), and the percent discount for units above the discount volume. Using normal spreadsheet formulas, users would have to set up three lookup formulas together with some logical tests to calculate the invoice amount.

Figure 1-16



The `InvoiceAmount()` function has three input parameters: `Product` is the name of the product, `Volume` is the number of units sold, and `Table` is the lookup table. The formula in cell C9, in Figure 1-16, defines the ranges to be used for each input parameter:

```
Function InvoiceAmount(Product, Volume, Table)
  'Find price in table
  Price = WorksheetFunction.VLookup(Product, Table, 2)

  'Find discount volume threshold
  DiscountVolume = WorksheetFunction.VLookup(Product, Table, 3)

  'Apply discount if volume above threshold
  If Volume > DiscountVolume Then
    'Calculate invoice with discount
    DiscountPct = WorksheetFunction.VLookup(Product, Table, 4)
    InvoiceAmount = Price * DiscountVolume + Price *
```

```
                    (1 - DiscountPct) * (Volume - DiscountVolume)
  Else
    'Calculate invoice without discount
    InvoiceAmount = Price * Volume
  End If
End Function
```

The range for the table is absolute so that the copies of the formula below cell C8 refer to the same range. The first calculation in the function uses the `VLookup` function to find the product in the lookup table and return the corresponding value from the second column of the lookup table, which it assigns to the variable `Price`.

> *If you want to use an Excel worksheet function in a VBA procedure, you need to tell VBA where to find it by preceding the function name with* `WorksheetFunction` *and a period. For compatibility with Excel 5 and 95, you can use* `Application` *instead of* `WorksheetFunction`*. Not all worksheet functions are available this way. In these cases, VBA has equivalent functions, or mathematical operators, to carry out the same calculations.*

In the next line of the function, the discount volume is found in the lookup table and assigned to the variable `DiscountVolume`. The `If` test on the next line compares the sales volume in `Volume` with `DiscountVolume`. If `Volume` is greater than `DiscountVolume`, the calculations following it, down to the `Else` statement, are carried out. Otherwise, the calculation after the `Else` is carried out.

If `Volume` is greater than `DiscountVolume`, the percent discount rate is found in the lookup table and assigned to the variable `DiscountPct`. The invoice amount is then calculated by applying the full price to the units up to `DiscountVolume` plus the discounted price for units above `DiscountVolume`. Note the use of the underscore character, preceded by a blank space, to indicate the continuation of the code on the next line.

The result is assigned to the name of the function, `InvoiceAmount`, so that the value will be returned to the worksheet cell. If `Volume` is not greater than `DiscountVolume`, the invoice amount is calculated by applying the price to the units sold, and the result is assigned to the name of the function.

Direct Reference to Ranges

When you define a UDF, it is possible to directly refer to worksheet ranges rather than through the input parameters of the UDF. This is illustrated in the following version of the `InvoiceAmount()` function:

```
Function InvoiceAmount2(Product, Volume)
  'Create object variable referring to table in worksheet
  Set Table = ThisWorkbook.Worksheets("Sheet2").Range("A2:D5")

  'Find price in table
  Price = WorksheetFunction.VLookup(Product, Table, 2)

  'Find discount volume threshold
  DiscountVolume = WorksheetFunction.VLookup(Product, Table, 3)

  'Apply discount if volume above threshold
  If Volume > DiscountVolume Then
    'Calculate invoice with discount
    DiscountPct = WorksheetFunction.VLookup(Product, Table, 4)
    InvoiceAmount2 = Price * DiscountVolume + Price *
                    (1 - DiscountPct) * (Volume - DiscountVolume)
  Else
    'Calculate invoice without discount
    InvoiceAmount2 = Price * Volume
  End If
  End Function
```

Note that `Table` is no longer an input parameter. Instead, the `Set` statement defines `Table` with a direct reference to the worksheet range. Although this method still works, the return value of the function will not be recalculated if you change a value in the lookup table. Excel does not realize that it needs to recalculate the function when a lookup table value changes, because it does not see that the table is used by the function.

Excel only recalculates a UDF when it sees its input parameters change. If you want to remove the lookup table from the function parameters and still have the UDF recalculate

automatically, you can declare the function to be volatile on the first line of the function, as shown here:

```
Function InvoiceAmount2(Product, Volume)
  Application.Volatile
  Set Table = ThisWorkbook.Worksheets("Sheet2").Range("A2:D5")
  …
```

However, you should be aware that this feature comes at a price. If a UDF is declared volatile, the UDF is recalculated every time any value changes in the worksheet. This can add a significant recalculation burden to the worksheet if the UDF is used in many cells.

What UDFs Cannot Do

A common mistake made by users is to attempt to create a worksheet function that changes the structure of the worksheet by, for example, copying a range of cells. Such attempts will fail. No error messages are produced because Excel simply ignores the offending code lines, so the reason for the failure is not obvious.

> **UDFs, used in worksheet cells, are not permitted to change the structure of the worksheet, meaning that a UDF cannot return a value to any other cell than the one it is used in, and it cannot change a physical characteristic of a cell, such as the font color or background pattern. In addition, UDFs cannot carry out actions such as copying or moving spreadsheet cells. They cannot even carry out some actions that imply a change of cursor location, such as an Edit ➪ Find. A UDF can call another function procedure, or even a sub procedure, but that procedure will be under the same restrictions as the UDF. It will still not be permitted to change the structure of the worksheet.**

A distinction is made (in Excel VBA) between UDFs that are used in worksheet cells and function procedures that are not connected with worksheet cells. As long as the original calling procedure was not a UDF in a worksheet cell, a function procedure can carry out any Excel action, just like a sub procedure.

It should also be noted that UDFs are not as efficient as the built-in Excel worksheet functions. If UDFs are used extensively in a workbook, recalculation time will be greater compared with a similar workbook using the same number of built-in functions.

The Excel Object Model

The Visual Basic for Applications programming language is common across all the Microsoft Office applications. In addition to Excel, you can use VBA in Word, Access, PowerPoint, and Outlook. Once you learn it, you can apply it to any of these. However, to work with an application, you need to learn about the objects it contains. In Word, you deal with documents, paragraphs, and words. In Access, you deal with databases, recordsets, and fields. In Excel, you deal with workbooks, worksheets, and ranges.

Unlike many programming languages, you don't have to create your own objects in Office VBA. Each application has a clearly defined set of objects that are arranged according to the relationships between them. This structure is referred to as the application's object model. This section is an introduction to the Excel object model, which is fully documented in Appendix A.

Objects

First up, this section covers a few basics about Object-Oriented Programming (OOP). This not a complete formal treatise on the subject, but it covers what you need to know to work with the objects in Excel.

OOP's basic premise is that you can describe everything known to us as objects. You and I are objects, the world is an object, and the universe is an object. In Excel, a workbook is an object, a worksheet is an object, and a range is an object. These objects are only a small sample of around two hundred object types available to us in Excel. Look at some examples of how to refer to `Range` objects in VBA code. One simple way to refer to cells B2:C4 is as follows:

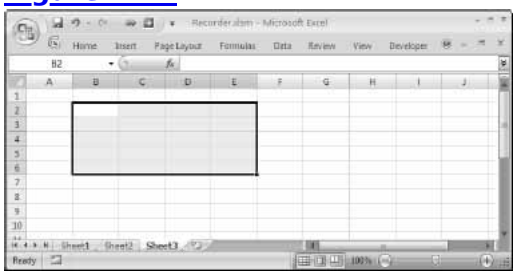
```
Range("B2:C4")
```

If you give the name `Data` to a range of cells, you can use that name in a similar way:

```
Range("Data")
```

There are also ways to refer to the currently active cell and selection using shortcuts. In Figure 1-17, `ActiveCell` refers to the B2 cell, and `Selection` refers to the range B2:E6. For more information on `ActiveCell` and `Selection`, see Chapter 3.

Figure 1-17



Collections

Many objects belong to collections. A city block is a collection of high-rise buildings. A high-rise building is a collection of floor objects. A floor is a collection of room objects. Collections are objects themselves—objects that contain other objects that are closely related. Collections and objects are often related in a hierarchical or tree structure.

Excel is an object itself, called the `Application` object. In the Excel `Application` object, there is a `Workbooks` collection that contains all the currently open `Workbook` objects. Each `Workbook` object has a `Worksheets` collection that contains the `Worksheet` objects in that workbook.

> **Note that you need to make a clear distinction between the plural `Worksheets` object, which is a collection, and the singular `Worksheet` object. They are quite different objects.**

If you want to refer to a member of a collection, you can refer to it by its position in the collection, as an index number starting with 1, or by its name, as quoted text. If you have opened just one workbook called `Data.xls`, you can refer to it by either of the following:

```
Workbooks(1)
Workbooks("Data.xls")
```

If you have three worksheets in the active workbook that have the names `North`, `East`, and `South`, in that order, you can refer to the second worksheet by either of the following:

```
Worksheets(2)
Worksheets("East")
```

If you want to refer to a worksheet called `DataInput` in a workbook called `Sales.xls`, and `Sales.xls` is not the active workbook, you must qualify the worksheet reference with the workbook reference, separating them with a period, as follows:

```
Workbooks("Sales.xls").Worksheets("DataInput")
```

When you refer to the B2 cell in `DataInput`, while another workbook is active, you use:

```
Workbooks("Sales.xls").Worksheets("DataInput").Range("B2")
```

The following section examines objects more closely and explains how you can manipulate them in VBA code. You need to be aware of two key characteristics of objects to do this.

They are the properties and methods associated with an object.

Properties

Properties are the physical characteristics of objects, and can be measured or quantified. You and I have a height property, an age property, a bank balance property, and a name property. Some of our properties can be changed fairly easily, such as our bank balance. Other properties are more difficult or impossible to change, such as our name and age.

A worksheet `Range` object has a `RowHeight` property and a `ColumnWidth` property. A `Workbook` object has a `Name` property, which contains its filename. Some properties can be changed easily, such as the `Range` object's `ColumnWidth` property, by assigning the property a new value. Other properties, such as the `Workbook` object's `Name` property, are read-only. You can't change the `Name` property by simply assigning a new value to it.

You refer to the property of an object by referring to the object, then the property, separated by a period. For example, to change the width of the column containing the active cell to 20 points, you would assign the value to the `ColumnWidth` property of the `ActiveCell` using:

```
ActiveCell.ColumnWidth = 20
```

To enter the name `Florence` into cell C10, you assign the name to the `Value` property of the `Range` object:

```
Range("C10").Value = "Florence"
```

If the `Range` object is not in the active worksheet in the active workbook, you need to be more specific:

```
Workbooks("Sales.xls").Worksheets("DataInput").Range("C10").Value = 10
```

> **VBA can do what is impossible to do manually. It can enter data into worksheets that are not visible on the screen. It can copy and move data without having to make the sheets involved active. Therefore, it is very seldom necessary to activate a specific workbook, worksheet, or range to manipulate data using VBA. The more you can avoid activating objects, the faster your code will run. Unfortunately, the macro recorder can only record what you do and uses activation extensively.**

In the previous examples, you have seen how to assign values to the properties of objects. You can also assign the property values of objects to variables or to other objects' properties. You can directly assign the column width of one cell to another cell on the active sheet, using:

```
Range("C1").ColumnWidth = Range("A1").ColumnWidth
```

You can assign the value in C1 in the active sheet to D10 in the sheet named `Sales`, in the active workbook, using:

```
Worksheets("Sales").Range("D10").Value = Range("C1").Value
```

You can assign the value of a property to a variable so it can be used in later code. This example stores the current value of cell M100, sets M100 to a new value, prints the auto-recalculated results, and sets M100 back to its original value:

```
OpeningStock = Range("M100").Value
Range("M100").Value = 100
ActiveSheet.PrintOut
Range("M100").Value = OpeningStock
```

Some properties are read-only, which means that you can't assign a value to them directly. Sometimes there is an indirect way. One example is the `Text` property of a `Range` object. You can assign a value to a cell using its `Value` property, and you can give the cell a number format using its `NumberFormat` property. The `Text` property of the cell gives you the formatted appearance of the cell. The following example displays `$12,345.60` in a Message box:

```
Range("B10").Value = 12345.6
Range("B10").NumberFormat = "$#,##0.00"
MsgBox Range("B10").Text
```

This is the only means by which you can set the value of the `Text` property.

Methods

Whereas properties are the quantifiable characteristics of objects, methods are the actions that can be performed by objects or on objects. If you have a linguistic bent, you might like to think of objects as nouns, properties as adjectives, and methods as verbs. Methods often change the properties of objects. I have a walking method that takes me from A to B, changing my location property. I have a spending method that reduces my bank balance property and a working method that increases my bank balance property. My dieting method reduces my weight property, temporarily.

A simple example of an Excel method is the `Select` method of the `Range` object. To refer to a method, as with properties, put the object first, add a period, and then add the method. The following selects cell G4:

```
Range("G4").Select
```

Another example of an Excel method is the `Copy` method of the `Range` object. The following copies the contents of range A1:B3 to the clipboard:

```
Range("A1:B3").Copy
```

Methods often have parameters that you can use to modify the way the method works. For example, you can use the `Paste` method of the `Worksheet` object to paste the contents of the clipboard into a worksheet, but if you do not specify where the data is to be pasted, it is inserted with its top-left corner in the active cell. This can be overridden with the `Destination` parameter (parameters are discussed later in this section):

```
ActiveSheet.Paste Destination:=Range("G4")
```

Note that the value of a parameter is specified using :=, not just =.

Often, Excel methods provide shortcuts. The previous examples of `Copy` and `Paste` can be carried out entirely by the `Copy` method:

```
Range("A1:B3").Copy Destination:=Range("G4")
```

This is far more efficient than the code produced by the macro recorder:

```
Range("A1:B3").Select
Selection.Copy
Range("G4").Select
ActiveSheet.Paste
```

Events

Another important concept in VBA is that objects can respond to events. A mouse click on a command button, a double-click on a cell, a recalculation of a worksheet, and the opening and closing of a workbook are examples of events.

All of the ActiveX controls can respond to events. These controls can be embedded in worksheets and in UserForms to enhance the functionality of those objects. Worksheets and workbooks can also respond to a wide range of events. If you want an object to respond to an event, enter VBA code into the appropriate event procedure for that object. The event procedure resides in the code module behind the `Workbook`, `Worksheet`, or `UserForm` object concerned.

For example, you might want to detect that a user has selected a new cell and highlight the cell's complete row and column. You can do this by entering code in the `Worksheet_SelectionChange()` event procedure:

> First activate the VBE window and double-click the worksheet in the Project Explorer. From the drop-down lists at the top of the worksheet code module, choose `Worksheet` and `SelectionChange`, and enter the following code:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
  Rows.Interior.ColorIndex = xlColorIndexNone
  Target.EntireColumn.Interior.ColorIndex = 36
```