Excel® 2007 VBA
Programmer's Reference

John Green
Stephen Bullen
Rob Bovey
Michael Alexander

ISBN: 978-0-470-04643-2

Chapter 1
Primer in Excel VBA

SKU: 9785CH0000062

Wiley Publishing, Inc.

Excel® 2007 VBA Programmer's Reference

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published simultaneously in Canada

ISBN: 978-0-470-04643-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Excel 2007 VBA programmer's reference / John Green ... [et al.].
p. cm.
Includes index.
ISBN 978-0-470-04643-2 (paper/website)
1. Microsoft Excel (Computer file) 2. Business—Computer programs. I. Green, John, 1945-
HF5548.4.M523E92988 2007
005.54—dc22

2007004976

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.

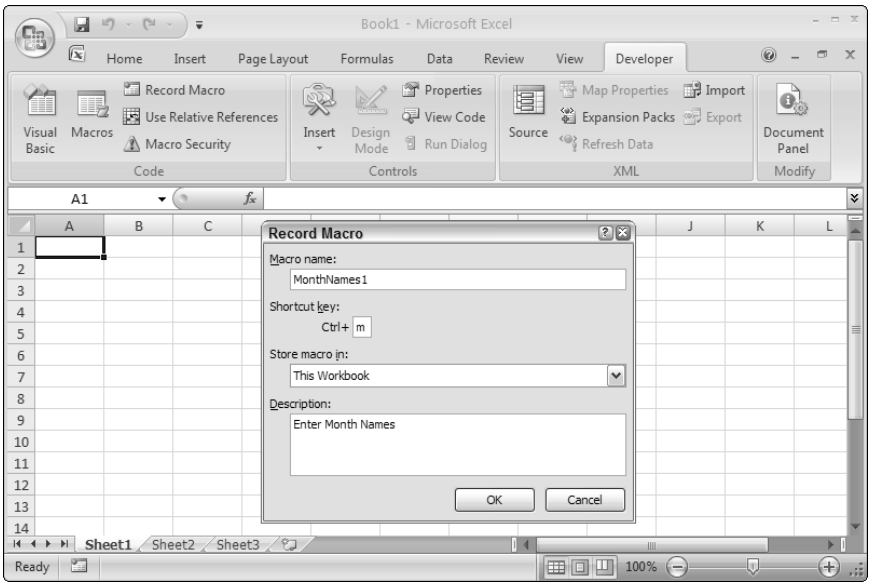


Figure 1-1

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 `MonthNames1`, 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.

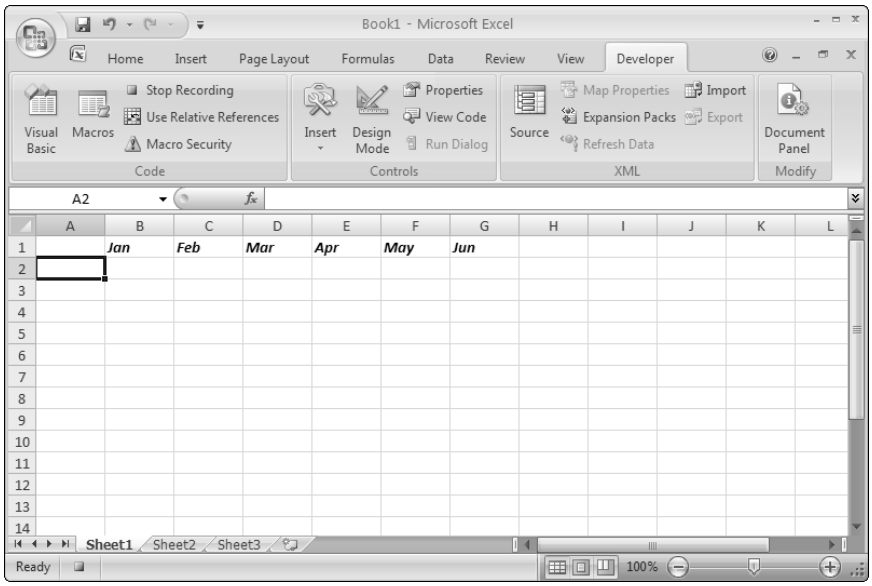


Figure 1-2

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.

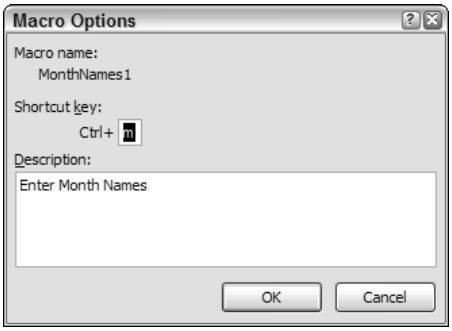


Figure 1-4

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 `MonthNames1`, 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 `MonthNames2`. 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 `MonthNames1`. 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.

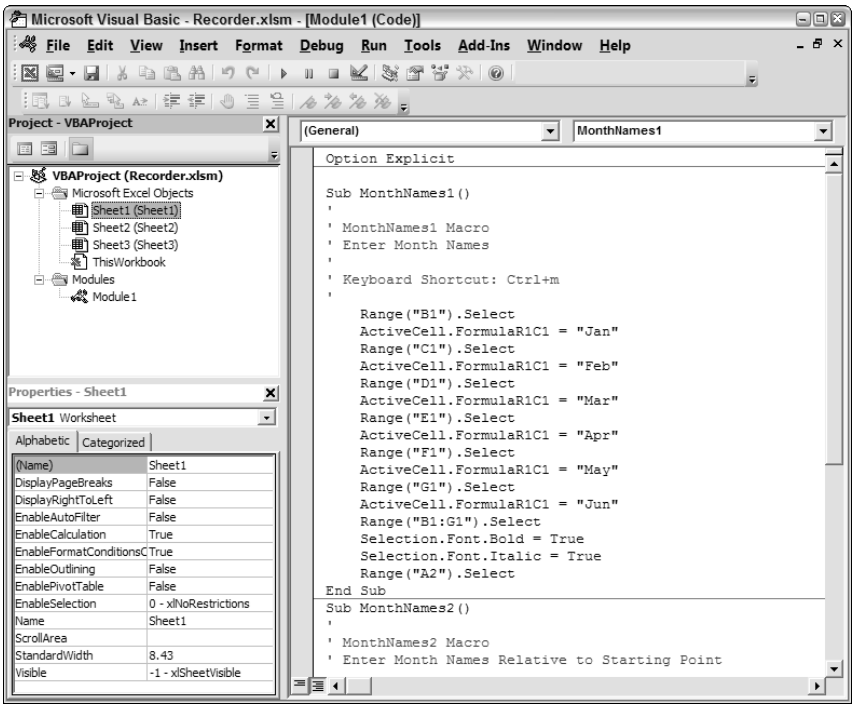


Figure 1-5

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 `MonthNames1` 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 `MonthNames1`, 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 `MonthNames1` and `MonthNames2`. `MonthNames1` 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.

You can use the Project Explorer to locate and activate the code modules in your project. You can double-click 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.

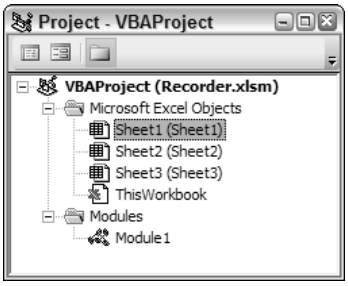


Figure 1-6

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.

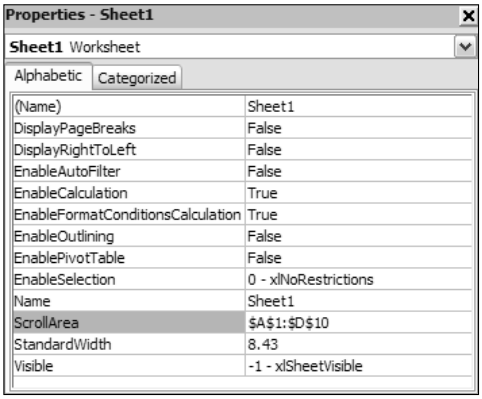


Figure 1-7

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.

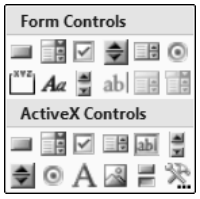


Figure 1-8

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.

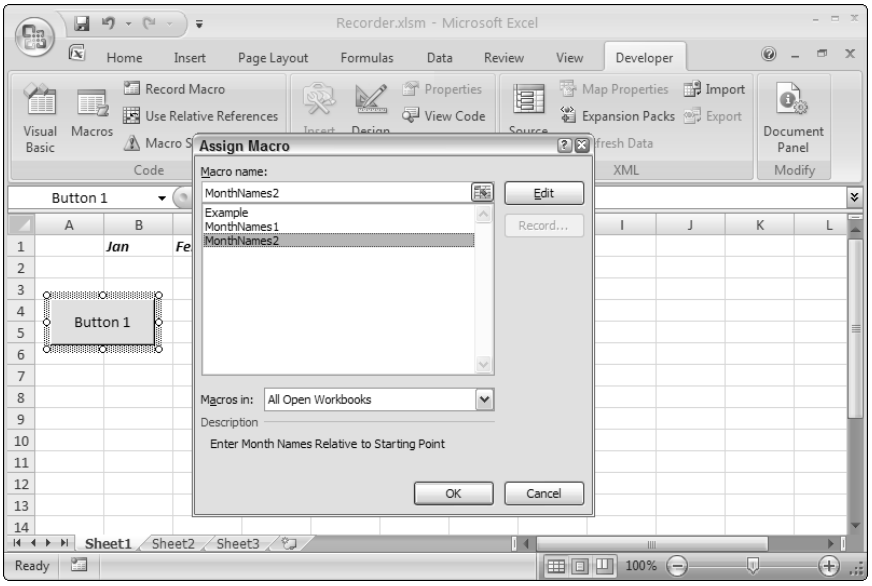


Figure 1-9

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.

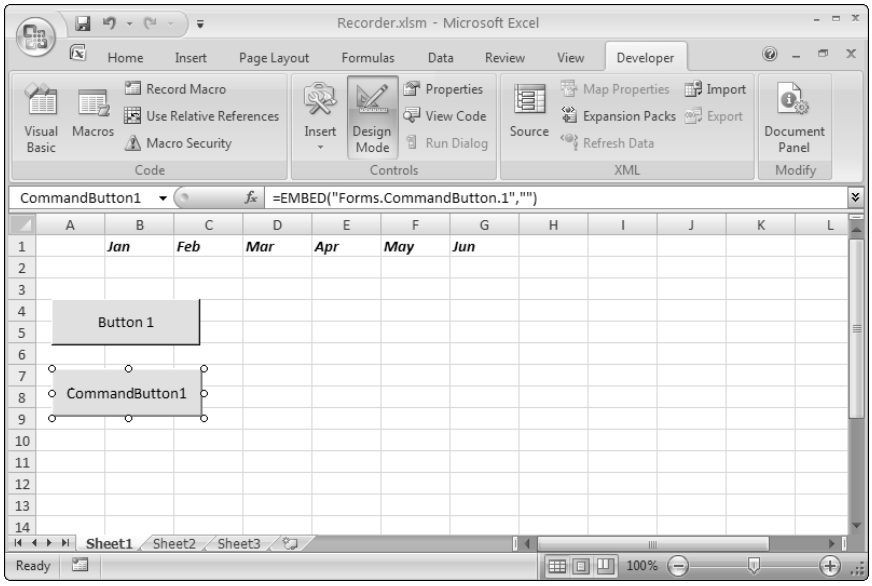


Figure 1-10

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.

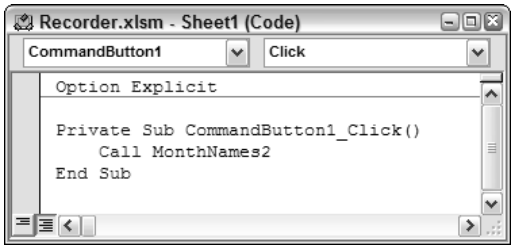


Figure 1-11

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.

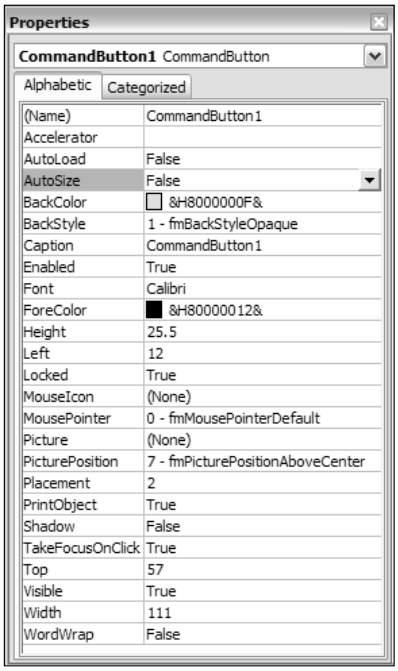


Figure 1-12

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.

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.

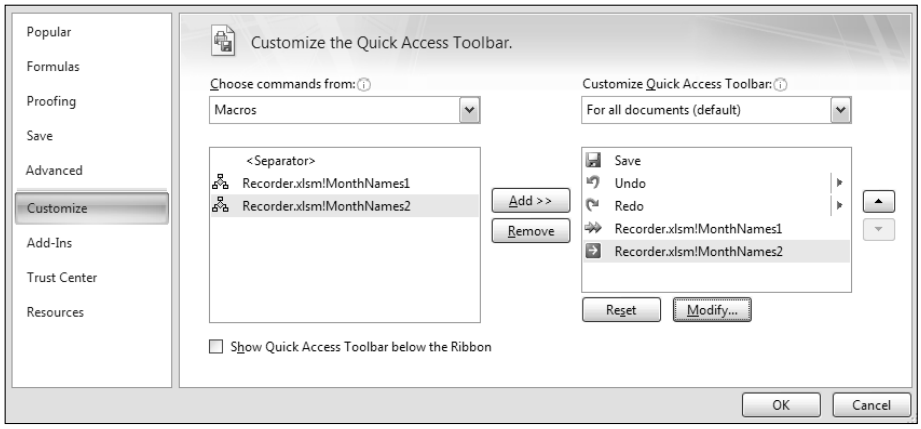


Figure 1-13

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.

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
```

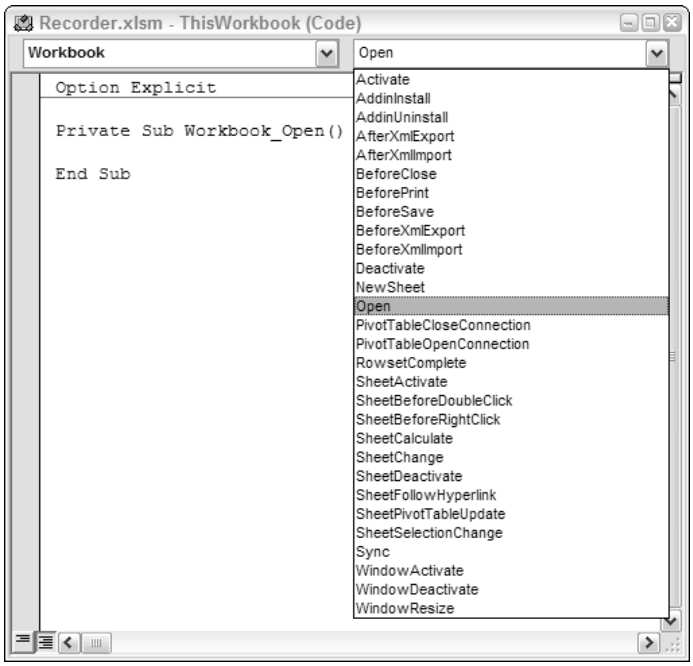


Figure 1-14

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.

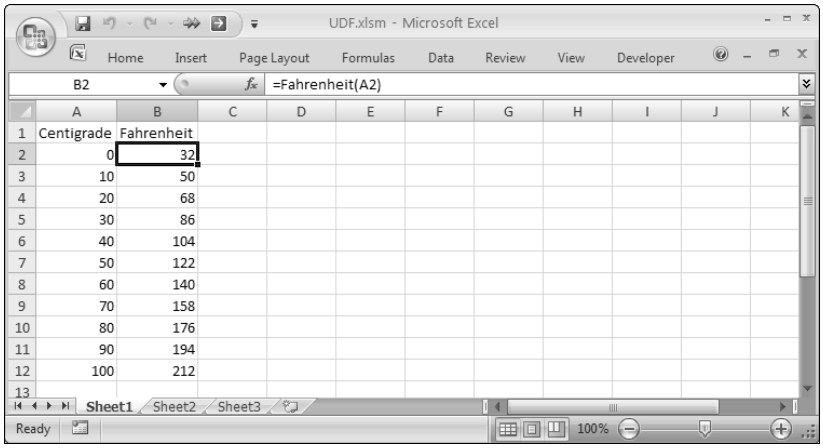


Figure 1-15

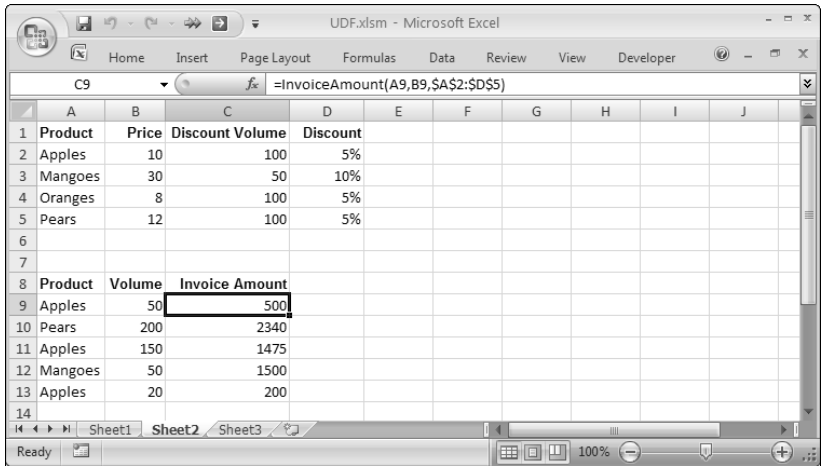
The formula has been copied into cells B3:B12.

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.



	A	B	C	D
1	**Product**	**Price**	**Discount Volume**	**Discount**
2	Apples	10	100	5%
3	Mangoes	30	50	10%
4	Oranges	8	100	5%
5	Pears	12	100	5%
6				
7				
8	**Product**	**Volume**	**Invoice Amount**	
9	Apples	50	500	
10	Pears	200	2340	
11	Apples	150	1475	
12	Mangoes	50	1500	
13	Apples	20	200	
14				

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.

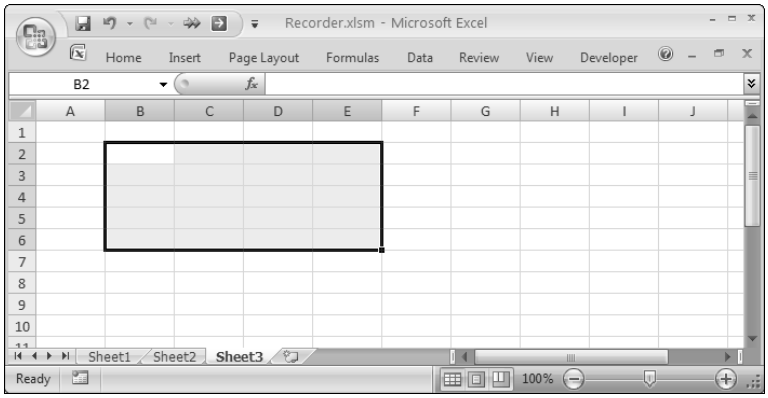


Figure 1-17

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:

1. First activate the VBE window and double-click the worksheet in the Project Explorer.
2. 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
  Target.EntireRow.Interior.ColorIndex = 36
End Sub
```

This event procedure runs every time the user selects a new cell, or block of cells. The parameter, `Target`, refers to the selected range as a `Range` object. The first statement sets the `ColorIndex` property of all the worksheets cells to no color, to remove any existing background color. The second and third statements set the entire columns and entire rows that intersect with the selected cells to a background color of pale yellow. This color can be different, depending on the color palette set up in your workbook.

The use of properties in this example is more complex than you have seen before. Now analyze the component parts. If you assume that `Target` is a `Range` object referring to cell B10, then the following code uses the `EntireColumn` property of the B10 `Range` object to refer to the entire B column, which is the range B1:B1048576, or B:B for short:

```
Target.EntireColumn.Interior.ColorIndex = 36
```

Similarly, the next line of code changes the color of row 10, which is the range A10:XFD10, or 10:10 for short:

```
Target.EntireRow.Interior.ColorIndex = 36
```

The `Interior` property of a `Range` object refers to an `Interior` object, which is the background of a range. Finally, set the `ColorIndex` property of the `Interior` object equal to the index number for the required color.

This code might appear to many to be far from intuitive. So how do you go about figuring out how to carry out a task involving an Excel object?

Getting Help

The easiest way to discover the required code to perform an operation is to use the macro recorder. The recorded code is likely to be inefficient, but it will indicate the objects required and the properties and methods involved. If you turn on the recorder to find out how to color the background of a cell, you will get something like the following:

```
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = 56
        .Color = 65535
        .TintAndShade = 0
        .PatternTintAndShade = 0
End With
```

This `With...End With` construction is discussed in more detail later in this chapter. It is equivalent to:

```
Selection.Interior.Pattern = xlSolid
Selection.Interior.PatternColorIndex = 56
Selection.Interior.Color = 65535
Selection.Interior.TintAndShade = 0
Selection.Interior.PatternTintAndShade = 0
```

The lines of code that specify `Pattern`, `TintAndShade`, and `PatternTintAndShade` are unnecessary, because they specify default values. The macro recorder is not sophisticated enough to know what the user does or doesn't want, so it includes everything. However, the recorded code provides the clues you need to get started. You only need to figure out how to change the `Range` object, `Selection`, into a complete row or complete column. If this can be done, it will be accomplished by using a property or method of the `Range` object.

The Object Browser

The Object Browser is a valuable tool for discovering the properties, methods, and events applicable to Excel objects. To display the Object Browser, you need to be in the VBE window. You can use View ➪ Object Browser, press F2, or click the Object Browser button on the Standard toolbar to see the window shown in Figure 1-18.

The objects are listed in the window with the title Classes. Objects are instances of classes. You can click in this window and type an r to get quickly to the `Range` object.

Alternatively, you can click in the search box, second from the top with the binoculars to its right, and type in `range`. When you press Enter or click the binoculars, you will see a list of items containing this text. When you click Range, under the Class heading in the Search Results window, Range will be highlighted in the Classes window below. This technique is handy when you are searching for information on a specific property, method, or event.

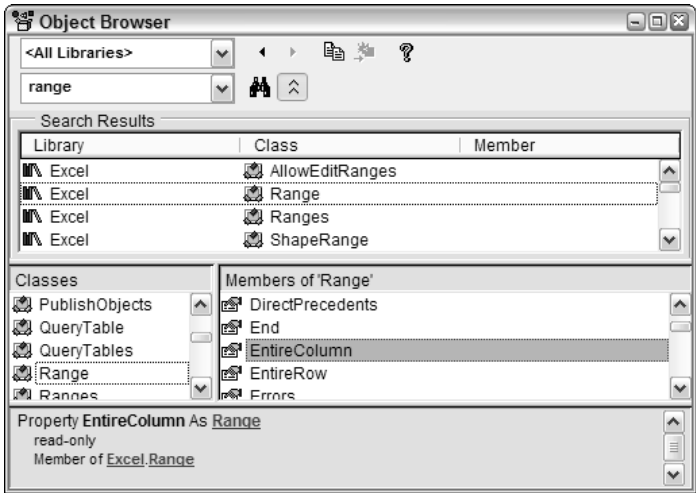


Figure 1-18

You now have a list of all the properties, methods, and events (if applicable) for this object, sorted alphabetically. If you right-click this list, you can choose Group Members to separate the properties, methods, and events, which makes it easier to read. If you scan through this list, you will see the `EntireColumn` and `EntireRow` properties, which look to be likely candidates for your requirements. To confirm this, select `EntireColumn` and click the question mark icon at the top of the Object Browser window to go to the window in Figure 1-19.

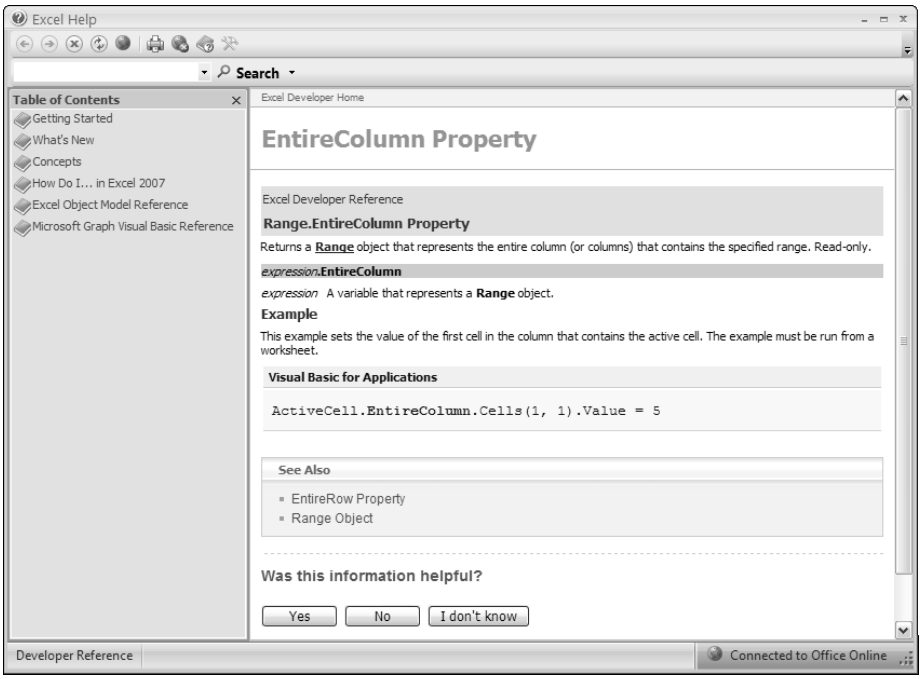


Figure 1-19

See Also can often lead to further information on related objects and methods. Now, all that remains to do is connect the properties you found and apply them to the right object.

Experimenting in the Immediate Window

If you want to experiment with code, you can use the VBE's Immediate window. Use View ➪ Immediate Window, press Ctrl+G, or click the Immediate Window button on the Debug toolbar to make the Immediate window visible. You can tile the Excel window and the VBE window so you can type commands into the Immediate window and see the effects in the Excel window, as shown in Figure 1-20.

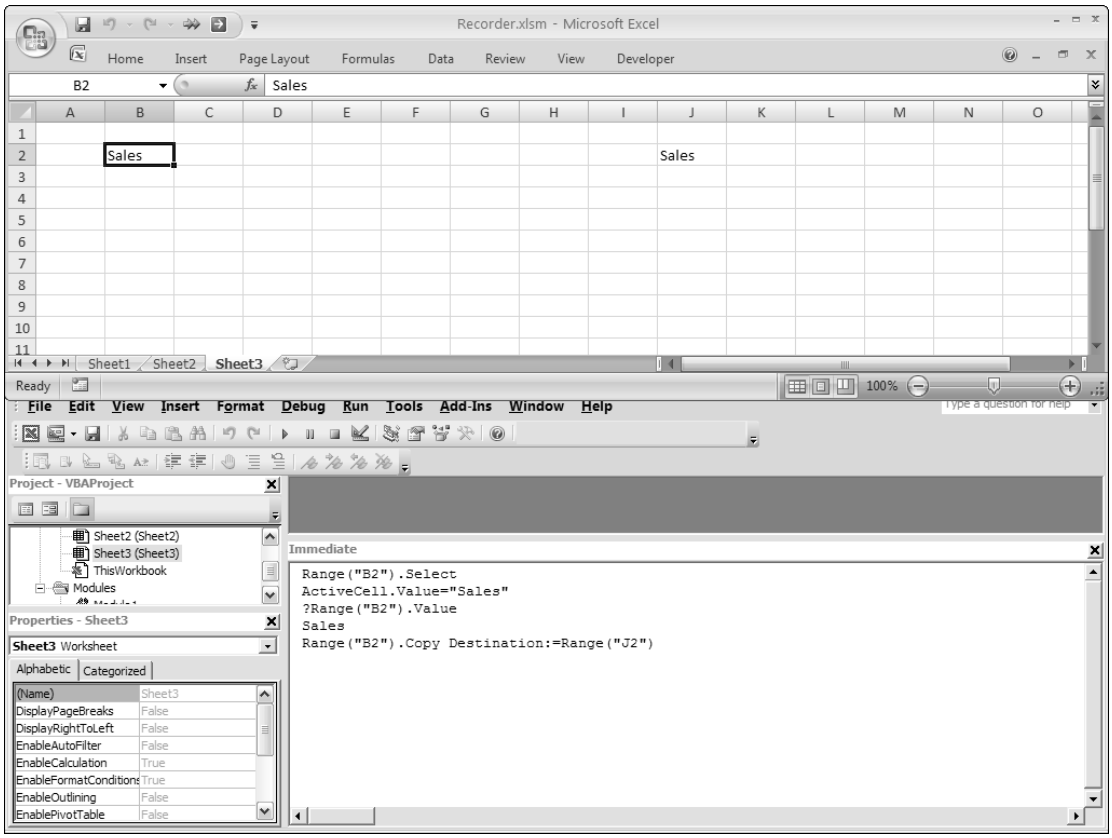


Figure 1-20

When a command is typed in and Enter is pressed, the command is immediately executed. To execute the same command again, click anywhere in the line with the command and press Enter again.

Here, the `Value` property of the `ActiveCell` object has been assigned the text "`Sales`". If you want to display a value, you precede the code with a question mark, which is a shortcut for `Print`:

```
?Range("B2").Value
```

This code has printed `Sales` on the next line of the Immediate window. The last command has copied the value in B2 to J2.

The VBA Language

In this section, you see the elements of the VBA language that are common to all versions of Visual Basic and the Microsoft Office applications. The section uses examples that employ the Excel object model, but the aim is to examine the common structures of the language. Many of these structures and concepts are common to other programming languages, although the syntax and keywords can vary. This section examines the following:

- Storing information in variables and arrays
- Decision-making in code
- Using loops
- Basic error-handling

Basic Input and Output

First, look at some simple communication techniques you can use to make your macros more flexible and useful. If you want to display a message, use the `MsgBox` function, which is useful if you want to display a warning message or ask a simple question.

In the first example, you want to make sure that the printer is switched on before a print operation. The following code generates the dialog box in Figure 1-21, giving the user a chance to check the printer. The macro pauses until the OK button is clicked:

```
MsgBox "Please make sure that the printer is switched on"
```

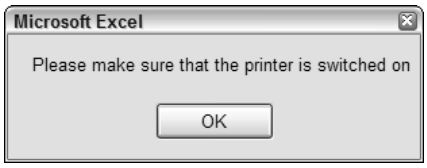


Figure 1-21

If you want to experiment, you can use the Immediate window to execute single lines of code. Alternatively, you can insert your code into a standard module in the VBE window. In this case, you need to include `Sub` and `End Sub` lines as follows:

```
Sub Test1()
  MsgBox "Please make sure that the printer is switched on"
End Sub
```

An easy way to execute a sub procedure is to click somewhere in the code to create an insertion point, then press F5.

`MsgBox` has many options that control the types of buttons and icons that appear in the dialog box. If you want to get help with this, or any VBA word, just click somewhere in the word and press the F1 key. The Help screen for the word will immediately appear. Among other details, you will see the input parameters accepted by the function:

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

Parameters in square brackets are optional, so only the `prompt` message is required. If you want to have a `title` at the top of the dialog box, you can specify the third parameter. There are two ways to specify parameter values: by position and by name.

Parameters Specified by Position

If you specify a parameter by position, you need to make sure that the parameters are entered in the correct order. You also need to include extra commas for missing parameters. The following code provides a title for the dialog box, specifying the title by position and producing the result shown in Figure 1-22:

```
MsgBox "Is the printer on?", , "Caution!"
```

Figure 1-22

Parameters Specified by Name

There are some advantages and some special considerations required when specifying parameters by name:

- You can enter them in any order and do not need to include extra commas with nothing between them to allow for undefined parameters.
- You do need to use `:=` rather than just `=` between the parameter name and the value, as already pointed out.

The following code generates the same dialog box as in Figure 1-22:

```
MsgBox Title:="Caution!", Prompt:="Is the printer on?"
```

Another advantage of specifying parameters by name is that the code is better documented. Anyone reading the code is more likely to understand it.

If you want more information on the `buttons` parameter, you will find a table of options in the help screen as follows:

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
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
vbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right-aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

Values 0 to 5 control the buttons that appear. A value of 4 gives Yes and No buttons, as shown in Figure 1-23:

```
MsgBox Prompt:="Delete this record?", Buttons:=4
```

Figure 1-23

Values 16 to 64 control the icons that appear; 32 gives a question mark icon. If you want both value 4 and value 32, add them to see the dialog box in Figure 1-24:

```
MsgBox Prompt:="Delete this record?", Buttons:=36
```

Figure 1-24

Constants

Specifying a `Buttons` value of 36 ensures that your code is indecipherable to all but the most battle-hardened programmer. This is why VBA provides the constants shown to the left of the button values in the help screen. Rather than specifying `Buttons` by numeric value, you can use the constants, which provide a better indication of the choice behind the value. The following code generates the same dialog box as the previous example:

```
MsgBox Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion
```

> **The VBE helps you as you type by providing a pop-up list of the appropriate constants after you type `Buttons:=`. Point to the first constant and press the plus key (+), and you will be prompted for the second constant. Choose the second and press the spacebar or Tab to finish the line. If there is another parameter to be specified, enter a comma rather than a space or a Tab.**

Constants are a special type of variable that do not change, if that makes sense. They are used to hold key data and, as you have seen, provide a way to write more understandable code. VBA has many built-in constants that are referred to as intrinsic constants. You can also define your own constants, as you will see later in this chapter.

Return Values

There is something missing from the previous examples of `MsgBox`. You are asking a question, but failing to capture the user's response to the question. That is because you have been treating `MsgBox` as a statement, rather than a function. This is perfectly legal, but you need to know some rules if you are to avoid syntax errors. You can capture the return value of the `MsgBox` function by assigning it to a variable.

However, if you try the following, you will get a syntax error:

```
Answer = MsgBox Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion
```

The error message, `Expected: End of Statement`, is not really very helpful. You can click the Help button on the error message to get a more detailed description of the error, but even then you might not understand the explanation.

Parentheses

The problem with the previous line of code is that there are no parentheses around the function arguments. It should read as follows:

```
Answer = MsgBox(Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion)
```

The general rule is that if you want to capture the return value of a function, you need to put any arguments in parentheses. If you don't want to use the return value, you should not use parentheses, as with the original examples of using `MsgBox`.

> *The parentheses rule also applies to methods used with objects. Many methods have return values that you can ignore or capture. See the section on object variables later in this chapter for an example.*

Now that you have captured the return value of `MsgBox`, how do you interpret it? Once again, the help screen provides the required information in the form of the following table of return values:

Constant	Value	Description
`vbOK`	1	OK
`vbCancel`	2	Cancel
`vbAbort`	3	Abort
`vbRetry`	4	Retry
`vbIgnore`	5	Ignore
`vbYes`	6	Yes
`vbNo`	7	No

If the Yes button is clicked, `MsgBox` returns a value of 6. You can use the constant `vbYes`, instead of the numeric value, in an `If` test:

```
Answer = MsgBox(Prompt:="Delete selected Row?", Buttons:=vbYesNo + vbQuestion)
If Answer = vbYes Then ActiveCell.EntireRow.Delete
...
```

InputBox

Another useful VBA function is `InputBox`, which allows you to get input data from a user in the form of text. The following code generates the dialog box shown in Figure 1-25:

```
UserName = InputBox(Prompt:="Please enter your name")
```

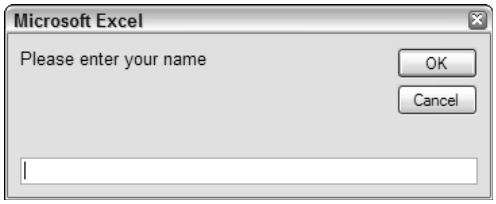


Figure 1-25

`InputBox` returns a text (string) result. Even if a numeric value is entered, the result is returned as text. If you click Cancel or OK without typing anything into the text box, `InputBox` returns a zero-length string. It is a good idea to test the result before proceeding so this situation can be handled. In the following example, the sub procedure does nothing if `Cancel` is clicked. The `Exit Sub` statement stops the procedure at that point. Otherwise, it places the entered data into cell B2:

```
Sub GetData()
   Sales = InputBox(Prompt:="Enter Target Sales")
   If Sales = "" Then Exit Sub
   Range("B2").Value = Sales
End Sub
```

In this code, the `If` test compares `Sales` with a zero-length string. There is nothing between the two double quote characters. Don't be tempted to put a blank space between the quotes.

> *There is a more powerful version of* `InputBox` *that is a method of the Excel Application object. It has the ability to restrict the type of data that you can enter. It is covered in Chapter 2.*

Calling Functions and Sub Procedures

When you develop an application, you should not attempt to place all your code in one large procedure. You should write small procedures that carry out specific tasks, and test each procedure independently. You can then write a master procedure that runs your task procedures. This approach makes the testing and debugging of the application much simpler, and also makes it easier to modify the application later.

The following code illustrates this modular approach, although in a practical application your procedures would have many more lines of code:

```
Sub Master()
   SalesData = GetInput("Enter Sales Data")
   If SalesData = False Then Exit Sub
   PostInput SalesData, "B3"
End Sub

Function GetInput(Message)
   Data = InputBox(Message)
   If Data = "" Then GetInput = False Else GetInput = Data
End Function

Sub PostInput(InputData, Target)
   Range(Target).Value = InputData
End Sub
```

`Master` uses the `GetInput` function and the `PostInput` sub procedure. `GetInput` has one input parameter, which passes the prompt message for the `InputBox` function and tests for a zero-length string in the response. A value of `False` is returned if this is found. Otherwise, `GetInput` returns the response.

`Master` tests the return value from `GetInput` and exits if it is `False`. Otherwise, `Master` calls `PostInput`, passing two values that define the data to be posted and the cell the data is to be posted to.

> **Note that sub procedures can accept input parameters, just like function procedures, if they are called from another procedure. You can't run a sub procedure with input parameters directly.**

Also note that, when calling `PostInput` and passing two parameters to it, `Master` does not place parentheses around the parameters. Because sub procedures do not generate a return value, you should not put parentheses around the arguments when one is called, except when using the `Call` statement that is discussed next.

When calling your own functions and subs, you can specify parameters by name, just as you can with built-in procedures. The following version of `Master` uses this technique:

```
Sub Master()
   SalesData = GetInput(Message:="Enter Sales Data")
   If SalesData = False Then Exit Sub

   PostInput Target:="B3", InputData:=SalesData

End Sub
```

The Call Statement

When running a sub procedure from another procedure, you can use the `Call` statement. There is no particular benefit in doing this; it is just an alternative to the previous method. `Master` can be modified as follows:

```
Sub Master()
  SalesData = GetInput("Enter Sales Data")
  If SalesData = False Then Exit Sub

  Call PostInput(SalesData, "B3")

End Sub
```

Note that if you use `Call`, you must put parentheses around the parameters passed to the called procedure, regardless of the fact that there is no return value from the procedure. You can also use `Call` with a function, but only if the return value is not used.

Parentheses and Argument Lists

As you have seen, the use of parentheses around arguments when calling procedures is a bit of a minefield, so the following sections summarize when to use them, at the risk of opening a can of worms and getting lost in mixed metaphors. Bear in mind that the same rules apply to argument lists of methods.

Without the Call Statement

Only place parentheses around the arguments when you are calling a function procedure and are also making use of the return value from the function procedure:

```
SalesData = GetInput("Enter Sales Data")
```

Don't place parentheses around the arguments when you are calling a function procedure and are not making use of the return value from the function procedure:

```
GetInput "Enter Sales Data"
```

Don't place parentheses around the arguments when you are calling a sub procedure:

```
PostInput SalesData, "B3"
```

An Important Subtlety Regarding Parentheses

The following is correct syntax and leads to untold confusion:

```
MsgBox ("Insert Disk")
```

It is not what it appears and it is not a negation of the parentheses rules. VBA has inserted a space between `MsgBox` and the left parenthesis, which it does not insert in the following:

```
Response = MsgBox("Insert Disk")
```

The extra space indicates that the parentheses are around the argument, not around the argument list. If you pass two input parameters, the following is not valid syntax:

```
MsgBox ("Insert Disk", vbExclamation)
```

The following is valid syntax:

```
MsgBox ("Insert Disk"), (vbExclamation)
```

It is fine to place parentheses around individual arguments, but not around the argument list. However, you might not get the result you expect.

Apologies if you are bored, but this is important stuff. It is more important when you get to refer to objects in parameter lists. Placing an object reference in parentheses causes VBA to convert the object reference to the object's default property. For example, `(Range("B1"))` is converted to the value in the B1 cell and is not a reference to a `Range` object. The following is valid syntax to copy A1 to B1:

```
Range("A1").Copy Range("B1")
```

The following is valid syntax but causes a run-time error:

```
Range("A1").Copy (Range("B1"))
```

With the Call Statement

If you use the `Call` statement, you must place parentheses around the arguments you pass to the called procedure:

```
Call PostInput(SalesData, "B3")
```

Because `Call` is of limited use, not being able to process a return value, and muddies the water with its own rules, it is preferable not to use it.

Variable Declaration

You have seen many examples of the use of variables for storing information. It is now time to discuss the rules for creating variable names, look at different types of variables, and talk about the best way to define variables.

> **Variable names can be constructed from letters and numbers and the underscore character. The name must start with a letter and can be up to 255 characters in length. It is a good idea to avoid using any special characters in variable names. To be on the safe side, you should only use the letters of the alphabet (upper- and lowercase), plus the numbers 0–9 and the underscore (_). Also, variable names can't be the same as VBA keywords, such as Sub and End, or VBA function names.**

So far you have been creating variables simply by using them. This is referred to as implicit variable declaration. Most computer languages require you to employ explicit variable declaration. This means that you must define the names of all the variables you are going to use before using them in code. VBA allows both types of declarations. If you want to declare a variable explicitly, do so using a `Dim` statement or one of its variations, which is discussed shortly. The following `Dim` statement declares a variable called `SalesData`:

```
Sub GetData()
   Dim SalesData
   SalesData = InputBox(Prompt:="Enter Target Sales")
   ...
```

Most users find implicit declaration easier than explicit declaration, but there are many advantages to being explicit. One advantage is the preservation of capitalization. More important advantages are discussed later in this chapter.

You might have noticed that if you enter VBA words, such as inputbox, in lowercase, they are automatically converted to VBA's standard capitalization when you move to the next line. This is a valuable form of feedback that tells you the word has been recognized as valid VBA code. It is a good idea to always type VBA words in lowercase and look for the change.

If you do not explicitly declare a variable name, you can get odd effects regarding its capitalization. Say you write the following code:

```
Sub GetData()
   SalesData = InputBox(Prompt:="Enter Target Sales")
   If salesdata = "" Then Exit Sub
   ...
```

You will find that when you press Enter at the end of line 3, the original occurrence of `SalesData` loses its capitalization and the procedure reads as follows:

```
Sub GetData()
   salesdata = InputBox(Prompt:="Enter Target Sales")
   If salesdata = "" Then Exit Sub
   ...
```

In fact, any time you edit the procedure and alter the capitalization of `salesdata`, the new version will be applied throughout the procedure. If you declare `SalesData` in a `Dim` statement, the capitalization you use on that line will prevail throughout the procedure. You can now type the variable name in lowercase in the body of the code and obtain confirmation that it has been correctly spelled as you move to a new line.

Option Explicit

There is a way to force explicit declaration in VBA. Place the statement `Option Explicit` in the declarations section of your module, which is at the very top of your module, before any procedures, as shown in Figure 1-26.

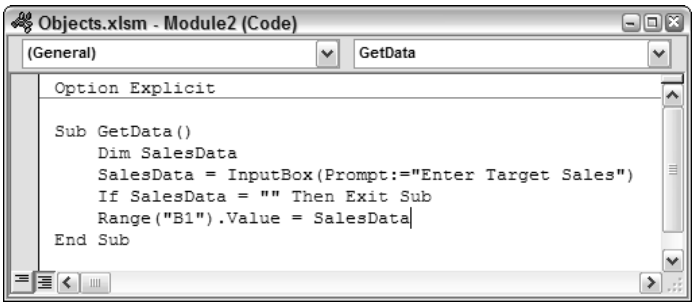


Figure 1-26

> `Option Explicit` **only applies to the module it appears in. Each module requiring explicit declaration of variables must repeat the statement in its declarations section.**

When you try to compile your module or run a procedure using explicit variable declaration, VBA will check for variables that have not been declared, highlight them, and show an error message. This has an enormous benefit. It picks up spelling mistakes, which are among the most common errors in programming. Consider the following version of `GetData`, where there is no `Option Explicit` at the top of the module and, therefore, implicit declaration is used:

```
Sub GetData()
   SalesData = InputBox(Prompt:="Enter Target Sales")
   If SaleData = "" Then Exit Sub
   Range("B2").Value = SalesData
End Sub
```

This code will never enter any data into cell B2. VBA happily accepts the misspelled `SaleData` in the `If` test as a new variable that is empty, and thus is considered to be a zero-length string for the purposes of the test. Consequently, the `Exit Sub` is always executed and the final line is never executed. This type of error, especially when embedded in a longer section of code, can be very difficult to see.

If you include `Option Explicit` in your declarations section, and `Dim SalesData` at the beginning of `GetData`, you will get an error message, `Variable not defined`, immediately after you attempt to run `GetData`. The undefined variable will be highlighted so that you can see exactly where the error is.

> **You can have** `Option Explicit` **automatically added to any new modules you create. In the VBE, use Tools➪Options and click the Editor tab. Check the box against Require Variable Declaration. This is a highly recommended option. Note that setting this option will not affect any existing modules, where you will need to insert** `Option Explicit` **manually.**

Scope and Lifetime of Variables

There are two important concepts associated with variables:

- The scope of a variable defines which procedures can use that variable
- The lifetime of a variable defines how long that variable retains the values assigned to it

The following procedure illustrates the lifetime of a variable:

```
Sub LifeTime()
  Dim Sales
  Sales = Sales + 1
  MsgBox Sales
End Sub
```

Every time `LifeTime` is run, it displays a value of one. This is because the variable `Sales` is only retained in memory until the end of the procedure. The memory `Sales` uses is released when the `End Sub` is reached. Next time `LifeTime` is run, `Sales` is re-created and treated as having a 0 value. The lifetime of `Sales` is the time taken to run the procedure. You can increase the lifetime of `Sales` by declaring it in a `Static` statement:

```
Sub LifeTime()
  Static Sales
  Sales = Sales + 1
  MsgBox Sales
End Sub
```

The lifetime of `Sales` is now extended to the time that the workbook is open. The more times `LifeTime` is run, the higher the value of `Sales` will become.

The following two procedures illustrate the scope of a variable:

```
Sub Scope1()
  Static Sales
  Sales = Sales + 1
  MsgBox Sales
End Sub

Sub Scope2()
  Static Sales
  Sales = Sales + 10
  MsgBox Sales
End Sub
```

The variable `Sales` in `Scope1` is not the same variable as the `Sales` in `Scope2`. Each time `Scope1` is executed, the value of its `Sales` will increase by one, independently of the value of `Sales` in `Scope2`. Similarly, the `Sales` in `Scope2` will increase by 10 with each execution of `Scope2`, independently of the value of `Sales` in `Scope1`. Any variable declared within a procedure has a scope that is confined to that procedure. A variable that is declared within a procedure is referred to as a procedure-level variable.

Variables can also be declared in the declarations section at the top of a module, as shown in the following version of the code:

```
Option Explicit
Dim Sales

Sub Scope1()
  Sales = Sales + 1
  MsgBox Sales
End Sub

Sub Scope2()
  Sales = Sales + 10
  MsgBox Sales
End Sub
```

`Scope1` and `Scope2` are now processing the same variable, `Sales`. A variable declared in the declarations section of a module is referred to as a module-level variable, and its scope is now the whole module. Therefore, it is visible to all the procedures in the module. Its lifetime is now the time that the workbook is open.

If a procedure in the module declares a variable with the same name as a module-level variable, the module-level variable will no longer be visible to that procedure. It will process its own procedure-level variable.

Module-level variables, declared in the declarations section of the module with a `Dim` statement, are not visible to other modules. If you want to share a variable between modules, you need to declare it as `Public` in the declarations section:

```
Public Sales
```

`Public` variables can also be made visible to other workbooks, or VBA projects. To accomplish this, a reference to the workbook containing the `Public` variable is created in the other workbook, using Tools ➪ References in the VBE.

Variable Type

Computers store different types of data in different ways. The way a number is stored is quite different from the way text, or a character string, is stored. Different categories of numbers are also stored in different ways. An integer (a whole number with no decimals) is stored differently from a number with decimals. Most computer languages require that you declare the type of data to be stored in a variable. VBA does not require this, but your code will be more efficient if you do declare variable types. It is also more likely that you will discover any problems that arise when data is converted from one type to another, if you have declared your variable types.

The following table has been taken directly from the VBA Help files. It defines the various data types available in VBA and their memory requirements. It also shows you the range of values that each type can handle:

Data type	Storage size	Range
`Byte`	1 byte	0 to 255
`Boolean`	2 bytes	`True` or `False`
`Integer`	2 bytes	-32,768 to 32,767
`Long` (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
`Single` (single-precision floating-point)	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
`Double` (double-precision floating-point)	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values

Data type	Storage size	Range
`Currency` (scaled integer)	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
`Decimal`	14 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; the smallest non-0 number is +/-0.0000000000000000000000000001
`Date`	8 bytes	January 1, 100 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`
`Variant` (with characters)	22 bytes + string length	Same range as for variable-length `String`
User-defined (using `Type`)	Number required by elements	The range of each element is the same as the range of its data type

If you do not declare a variable's type, it defaults to the `Variant` type. `Variants` take up more memory than any other type because each `Variant` has to carry information with it that tells VBA what type of data it is currently storing, as well as store the data itself.

`Variants` use more computer overhead when they are processed. VBA has to figure out what types it is dealing with and whether it needs to convert between types in order to process the number. If maximum processing speed is required for your application, you should declare your variable types, taking advantage of those types that use less memory when you can. For example, if you know your numbers will be whole numbers in the range of -32000 to +32000, you would use an `Integer` type.

Declaring Variable Type

You can declare a variable's type on a `Dim` statement, or related declaration statements such as `Public`. The following declares `Sales` to be a double precision floating-point number:

```
Dim Sales As Double
```

You can declare more than one variable on a `Dim`:

```
Dim SalesData As Double, Index As Integer, StartDate As Date
```

The following can be a trap:

```
Dim Col, Row, Sheet As Integer
```

Many users assume that this declares each variable to be `Integer`. This is not true. `Col` and `Row` are `Variant` because they have not been given a type. To declare all three as `Integer`, the line should be as follows:

```
Dim Col As Integer, Row As Integer, Sheet As Integer
```

Declaring Function and Parameter Types

If you have input parameters for sub procedures or function procedures, you can define each parameter type in the first line of the procedure as follows:

```
Function IsHoliday(WhichDay As Date)

Sub Marine(CrewSize As Integer, FuelCapacity As Double)
```

You can also declare the return value type for a function. The following example is for a function that returns a value of `True` or `False`:

```
Function IsHoliday(WhichDay As Date) As Boolean
```

Constants

You have seen that many intrinsic constants are built into VBA, such as `vbYes` and `vbNo`, discussed previously. You can also define your own constants. Constants are handy for holding numbers or pieces of text that do not change while your code is running, but that you want to use repeatedly in calculations and messages. Constants are declared using the `Const` keyword, as follows:

```
Const Pi = 3.14159265358979
```

You can include the constant's type in the declaration:

```
Const Version As String = "Release 3.9a"
```

Constants follow the same rules regarding scope as variables. If you declare a constant within a procedure, it will be local to that procedure. If you declare it in the declarations section of a module, it will be available to all procedures in the module. If you want to make it available to all modules, you can declare it to be `Public` as follows:

```
Public Const Error666 As String = "You can't do that"
```

Variable Naming Conventions

You can call your variables and user-defined functions anything you want, except where there is a clash with VBA keywords and function names. However, many programmers adopt a system whereby the variable or object type is included, in abbreviated form, in the variable name, usually as a prefix, so instead of declaring:

```
Dim SalesData As Double
```

you can use:

```
Dim dSalesData As Double
```

Wherever `dSalesData` appears in your code, you will be reminded that the variable is of type `Double`. Alternatively, you could use this line of code:

```
Dim dblSalesData As Double
```

For the sake of simplicity, this approach has not been used so far in this chapter, but from here onward, the examples will use a system to create variable names. This is the convention used in this book:

- One-letter prefixes for the common data types:

```
Dim iColumn As Integer
Dim lRow As Long
Dim dProduct As Double
Dim sName As String
Dim vValue As Variant
Dim bChoice As Boolean
```

- Two- or three-letter prefixes for object types:

```
Dim objExcel As Object
Dim rngData As Range
Dim wkbSales As Workbook
```

In addition to these characters, a lowercase `a` will be inserted in front of array variables, which are discussed later in this chapter. If the variable is a module-level variable, it will also have a lowercase `m` placed in front of it. If it is a public variable, it will have a lowercase `g` (for global) placed in front of it. For example, `malEffect` would be a module-level array variable containing long integer values.

Object Variables

The variables you have seen so far have held data such as numbers and text. You can also create object variables to refer to objects such as worksheets and ranges. The `Set` statement is used to assign an object reference to an object variable. Object variables should also be declared and assigned a type as with normal variables. If you don't know the type, you can use the generic term `Object` as the type:

```
Dim objWorkbook As Object
Set objWorkbook = ThisWorkbook
MsgBox objWorkbook.Name
```

It is more efficient to use the specific object type if you can. The following code creates an object variable `rng`, referring to cell B10 in Sheet1, in the same workbook as the code. It then assigns values to the object and the cell above:

```
Sub ObjectVariable()
  Dim rng As Range
  Set rng = ThisWorkbook.Worksheets("Sheet1").Range("C10")
  rng.Value = InputBox("Enter Sales for January")
  rng.Offset(-1, 0).Value = "January Sales"
End Sub
```

If you are going to refer to the same object more than once, it is more efficient to create an object variable than to keep repeating a lengthy specification of the object. It also makes code easier to read and write.

Object variables can also be very useful for capturing the return values of some methods, particularly when you are creating new instances of an object. For example, with either the `Workbooks` object or the `Worksheets` object, the `Add` method returns a reference to the new object. This reference can be assigned to an object variable so that you can easily refer to the new object in later code:

```
Sub NewWorkbook()
  Dim wkb As Workbook, wks As Worksheet

  Set wkb = Workbooks.Add
  Set wks = wkb.Worksheets.Add(After:=wkb.Sheets(wkb.Sheets.Count))
  wks.Name = "January"
  wks.Range("A1").Value = "Sales Data"
  wkb.SaveAs Filename:="JanSales.xlsx"
End Sub
```

This example creates a new empty workbook and assigns a reference to it to the object variable `wkb`. A new worksheet is added to the workbook, after any existing sheets, and a reference to the new worksheet is assigned to the object variable `wks`. The name on the tab at the bottom of the worksheet is then changed to January, and the heading Sales Data is placed in cell A1. Finally, the new workbook is saved as `JanSales.xlsx`.

Note that the parameter after the `Worksheets.Add` is in parentheses. Because you are assigning the return value of the `Add` method to the object variable, any parameters must be in parentheses. If the return value of the `Add` method were ignored, the statement would be without parentheses, as follows:

```
wkb.Worksheets.Add After:=wkb.Sheets(wkb.Sheets.Count)
```

With...End With

Object variables provide a useful way to refer to objects in shorthand, and are also more efficiently processed by VBA than fully qualified object strings. Another way to reduce the amount of code you write, and also increase processing efficiency, is to use a `With...End With` structure. The final example in the previous section could be rewritten as follows:

```
With wkb
  .Worksheets.Add After:=.Sheets(.Sheets.Count)
End With
```

VBA knows that anything starting with a period is a property or a method of the object following the `With`. You can rewrite the entire `NewWorkbook` procedure to eliminate the `wkb` object variable, as follows:

```
Sub NewWorkbook()
  Dim wks As Worksheet
  With Workbooks.Add
    Set wks = .Worksheets.Add(After:=.Sheets(.Sheets.Count))
    wks.Name = "January"
    wks.Range("A1").Value = "Sales Data"
    .SaveAs Filename:="JanSales.xlsx"
  End With
End Sub
```

You can take this a step further and eliminate the `wks` object variable:

```
Sub NewWorkbook()
  With Workbooks.Add
    With .Worksheets.Add(After:=.Sheets(.Sheets.Count))
      .Name = "January"
      .Range("A1").Value = "Sales Data"
    End With
    .SaveAs Filename:="JanSales.xlsx"
  End With
End Sub
```

If you find this confusing, you can compromise with a combination of object variables and `With...End With`:

```
Sub NewWorkbook()
  Dim wkb As Workbook, wks As Worksheet

  Set wkb = Workbooks.Add
  With wkb
    Set wks = .Worksheets.Add(After:=.Sheets(.Sheets.Count))
    With wks
      .Name = "January"
      .Range("A1").Value = "Sales Data"
    End With
    .SaveAs Filename:="JanSales.xlsx"
  End With
End Sub
```

`With...End With` is useful when references to an object are repeated in a small section of code.

Making Decisions

VBA provides two main structures for making decisions and carrying out alternative processing, represented by the `If` and `Select Case` statements. `If` is the more flexible one, but `Select Case` is better when you are testing a single variable.

If Statements

`If` comes in three forms: the `IIf` function, the one-line `If` statement, and the block `If` structure. The following `dTax` function uses the `IIf` (Immediate `If`) function:

```
Function dTax(dProfitBeforeTax As Double) As Double
  dTax = IIf(dProfitBeforeTax > 0, 0.3 * dProfitBeforeTax, 0)
End Function
```

`IIf` is similar to the Excel worksheet `IF` function. It has three input arguments: the first is a logical test, the second is an expression that is evaluated if the test is true, and the third is an expression that is evaluated if the test is false.

In this example, the `IIf` function tests that the `dProfitBeforeTax` value is greater than 0. If the test is true, `IIf` calculates 30% of `dProfitBeforeTax`. If the test is false, `IIf` calculates 0. The calculated `IIf` value is then assigned to the return value of the `Tax` function. The `Tax` function can be rewritten using the single-line `If` statement as follows:

```
Function dTax(dProfitBeforeTax As Double) As Double
  If dProfitBeforeTax > 0 Then dTax = 0.3 * dProfitBeforeTax Else dTax = 0
End Function
```

One difference between `IIf` and the single-line `If` is that the `Else` section of the single-line `If` is optional. The third parameter of the `IIf` function must be defined. In VBA, it is often useful to omit the `Else`:

```
    If dProfitBeforeTax < 0 Then MsgBox "A Loss has occurred", , "Warning"
```

Another difference is that, whereas `IIf` can only return a value to a single variable, the single-line `If` can assign values to different variables:

```
    If iJohnsScore > iMarysScore Then iJohn = iJohn + 1 Else iMary = iMary + 1
```

Block If

If you want to carry out more than one action when a test is `true`, you can use a block `If` structure, as follows:

```
    If iJohnsScore > iMarysScore Then
      iJohn = iJohn + 1
      iMary = iMary - 1
    End If
```

Using a block `If`, you must not include any code after the `Then`, on the same line. You can have as many lines after the test as required, and you must terminate the scope of the block `If` with an `End If` statement. A block `If` can also have an `Else` section, as follows:

```
    If iJohnsScore > iMarysScore Then
      iJohn = iJohn + 1
      iMary = iMary - 1
    Else
      iJohn = iJohn - 1
      iMary = iMary + 1
    End If
```

A block `If` can also have as many `ElseIf` sections as required:

```
    If iJohnsScore > iMarysScore Then
      iJohn = iJohn + 1
      iMary = iMary - 1
    ElseIf iJohnsScore < iMarysScore Then
      iJohn = iJohn - 1
      iMary = iMary + 1
    Else
      iJohn = iJohn + 1
      iMary = iMary + 1
    End If
```

When you have a block `If` followed by one or more `ElseIf` tests, VBA keeps testing until it finds a true section. It executes the code for that section and then proceeds directly to the statement following the `End If`. If no test is true, the `Else` section is executed.

A block `If` does nothing when all tests are false and the `Else` section is missing. Block `If` tests can be nested, one inside the other. You should make use of indenting to show the scope of each block. This is vital — you can get into an awful muddle with the nesting of `If` blocks within other `If` blocks, and `If` blocks within `Else` blocks, and so on. If code is unindented, it isn't easy, in a long series of nested `If` tests, to match each `End If` with each `If`:

```
    If Not ThisWorkbook.Saved Then
      lAnswer = MsgBox("Do you want to save your changes", vbQuestion + _
                                                                  vbYesNo)
      If lAnswer = vbYes Then
        ThisWorkbook.Save
        MsgBox ThisWorkbook.Name & " has been saved"
      End If
    End If
```

This code uses the `Saved` property of the `Workbook` object containing the code to see if the workbook has been saved since changes were last made to it. If changes have not been saved, the user is asked if they want to save changes. If the answer is yes, the inner block `If` saves the workbook and informs the user.

Select Case

The following block `If` is testing the same variable value in each section:

```
Function vPrice(sProduct As String) As Variant
  If sProduct = "Apples" Then
    vPrice = 12.5
  ElseIf sProduct = "Oranges" Then
    vPrice = 15
  ElseIf sProduct = "Pears" Then
    vPrice = 18
  ElseIf sProduct = "Mangoes" Then
    vPrice = 25
  Else
    vPrice = CVErr(xlErrNA)
  End If
End Function
```

If `sProduct` is not found, the `vPrice` function returns an Excel error value of `#NA`. Note that `vPrice` is declared as a `Variant` so it can handle the error value as well as numeric values. For a situation like this, `Select Case` is a more elegant construction. It looks like this:

```
Function vPrice(sProduct As String) As Variant
  Select Case sProduct
    Case "Apples"
      vPrice = 12.5
    Case "Oranges"
      vPrice = 15
    Case "Pears"
      vPrice = 18
    Case "Mangoes"
      vPrice = 25
    Case Else
      vPrice = CVErr(xlErrNA)
  End Select
End Function
```

If you have only one statement per case, the following format works quite well. You can place multiple statements on a single line by placing a colon between statements:

```
Function vPrice(sProduct As String) As Variant
  Select Case sProduct
    Case "Apples":  vPrice = 12.5
    Case "Oranges": vPrice = 15
    Case "Pears":   vPrice = 18
    Case "Mangoes": vPrice = 25
    Case Else:      vPrice = CVErr(xlErrNA)
  End Select
End Function
```

`Select Case` can also handle ranges of numbers or text, as well as comparisons using the keyword `Is`. The following example calculates a fare of 0 for infants up to 3 years old and anyone older than 65, with two ranges between. Negative ages generate an error:

```
Function vFare(iAge As Integer) As Variant
  Select Case iAge
    Case 0 To 3, Is > 65
      vFare = 0
    Case 4 To 15
      vFare = 10
    Case 16 To 65
      vFare = 20
    Case Else
      vFare = CVErr(xlErrNA)
  End Select
End Function
```

Looping

All computer languages provide a mechanism for repeating the same, or similar, operations in an efficient way. VBA has two main structures that allow you to loop through the same code over and over again. They are the `Do...Loop` and the `For...Next` loop.

The `Do...Loop` is for those situations where the loop will be terminated when a logical condition applies, such as reaching the end of your data. The `For...Next` loop is for situations where you can predict in advance how many times you want to loop, such as when you want to enter expenses for the 10 people in your department.

VBA also has an interesting variation on the `For...Next` loop that is used to process all the objects in a collection — the `For Each...Next` loop. You can use it to process all the cells in a range or all the sheets in a workbook, for example.

Do...Loop

To illustrate the use of a `Do...Loop`, construct a sub procedure to shade every second line of a worksheet, as shown in Figure 1-27, to make it more readable. You want to apply the macro to different report sheets with different numbers of products, so the macro will need to test each cell in the A column until it gets to an empty cell to determine when to stop.

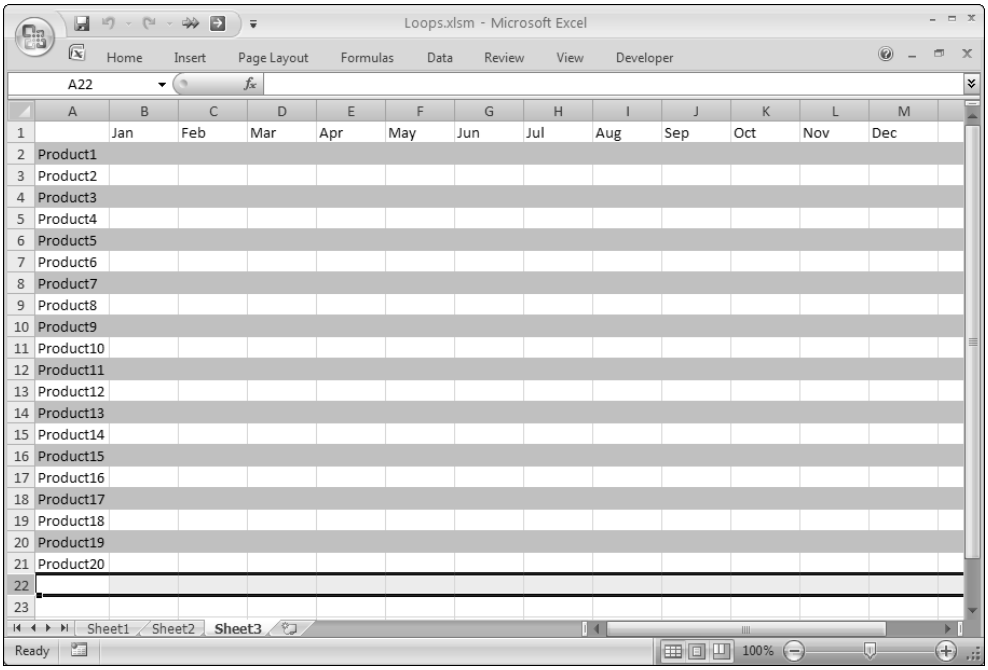


Figure 1-27

The first macro will select every other row and apply the formatting:

```
Sub ShadeEverySecondRow()
  Range("A2").EntireRow.Select
  Do While ActiveCell.Value <> ""
    Selection.Interior.ColorIndex = 15
    ActiveCell.Offset(2, 0).EntireRow.Select
  Loop
End Sub
```

`ShadeEverySecondRow` begins by selecting row 2 in its entirety. When you select an entire row, the left-most cell (in column A) becomes the active cell. The code between the `Do` and `Loop` statements is then repeated `While` the value property of the active cell is not a zero-length string, that is, the active cell is not empty. In the loop, the macro sets the interior color index of the selected cells to 15, which is gray. Then the macro selects the entire row, two rows under the active cell. When a row is selected that has an empty cell in column A, the `While` condition is no longer true and the loop terminates.

You can make `ShadeEverySecondRow` run faster by avoiding selecting. It is seldom necessary to select cells in VBA, but you are led into this way of doing things because that's the way you do it manually, and that's what you get from the macro recorder.

The following version of `ShadeEverySecondRow` does not select cells, and it runs considerably faster. It sets up an index `lRow`, which indicates the row of the worksheet and is initially assigned a value of 2. The `Cells` property of the worksheet allows you to refer to cells by row number and column number, so

when the loop starts, `Cells(lRow,1)` refers to cell A2. Each time around the loop, `i` is increased by two. You can, therefore, change any reference to the active cell to a `Cells(lRow,1)` reference and apply the `EntireRow` property to `Cells(lRow,1)` to refer to the complete row:

```
Sub ShadeEverySecondRow()
  Dim lRow As Long
  lRow= 2
  Do Until IsEmpty(Cells(lRow, 1))
    Cells(lRow, 1).EntireRow.Interior.ColorIndex = 15
    lRow= lRow+ 2
  Loop
End Sub
```

To illustrate some alternatives, two more changes have been made on the `Do` statement line in the previous code. Either `While` or `Until` can be used after the `Do`, so the test has been changed to an `Until` and you have used the VBA `IsEmpty` function to test for an empty cell.

> **The `IsEmpty` function is the best way to test that a cell is empty. If you use `If Cells (lRow,1) = ""`, the test will be true for a formula that calculates a zero-length string.**

It is also possible to exit a loop using a test within the loop and the `Exit Do` statement, as follows, which also shows another way to refer to entire rows:

```
Sub ShadeEverySecondRow()
  Dim lRow as Long
  lRow= 0
  Do
    lRow= lRow+ 2
    If IsEmpty(Cells(lRow, 1)) Then Exit Do
    Rows(lRow).Interior.ColorIndex = 15
  Loop
End Sub
```

Yet another alternative is to place the `While` or `Until` on the `Loop` statement line. This ensures that the code in the loop is executed at least once. When the test is on the `Do` line, it is possible that the test will be false to start with, and the loop will be skipped.

Sometimes, it makes more sense if the test is on the last line of the loop. In the following example, it seems more sensible to test `sPassWord` after getting input from the user, although the code would still work if the `Until` statement were placed on the `Do` line:

```
Sub GetPassword()
  Dim sPassWord As String, i As Integer
  i = 0
  Do
    i = i + 1
    If i > 3 Then
      MsgBox "Sorry, Only three tries"
      Exit Sub
    End If
```

```
    sPassWord = InputBox("Enter Password")
    Loop Until sPassWord = "XXX"
  MsgBox "Welcome"
End Sub
```

`GetPassword` loops until the password `XXX` is supplied, or the number of times around the loop exceeds three.

For...Next Loop

The `For...Next` loop differs from the `Do...Loop` in two ways. It has a built-in counter that is automatically incremented each time the loop is executed, and it is designed to execute until the counter exceeds a predefined value, rather than depending on a user-specified logical test. The following example places the full file path and name of the workbook into the center footer for each worksheet in the active workbook:

```
Sub FilePathInFooter()
  Dim i As Integer, sFilePath As String

  sFilePath = ActiveWorkbook.FullName
  For i = 1 To Worksheets.Count Step 1
    Worksheets(i).PageSetup.CenterFooter = sFilePath
  Next i
End Sub
```

Versions of Excel prior to Excel 2002 do not have an option to automatically include the full file path in a custom header or footer, so this macro inserts the information as text. It begins by assigning the `FullName` property of the active workbook to the variable `sFilePath`. The loop starts with the `For` statement and loops on the `Next` statement. `i` is used as a counter, starting at 1 and finishing when `i` exceeds `Worksheets.Count`, which uses the `Count` property of the `Worksheets` collection to determine how many worksheets there are in the active workbook.

The `Step` option defines the amount that `i` will be increased each time around the loop. `Step 1` could be left out of this example, because a step of 1 is the default value. In the loop, `i` is used as an index to the `Worksheets` collection to specify each individual `Worksheet` object. The `PageSetup` property of the `Worksheet` object refers to the `PageSetup` object in that worksheet, so that the `CenterFooter` property of the `PageSetup` object can be assigned the `sFilePath` text.

The following example shows how you can step backwards. It takes a complete file path and strips out the filename, excluding the file extension. The example uses the `FullName` property of the active workbook as input, but the same code could be used with any file path. It starts at the last character in the file path and steps backwards until it finds the period between the filename and its extension, and then the backslash character before the filename. It then extracts the characters between the two:

```
Sub GetFileName()
  Dim iBackSlash As Integer, iPoint As Integer
  Dim sFilePath As String, sFileName As String
  Dim i As Integer

  sFilePath = ActiveWorkbook.FullName
  For i = Len(sFilePath) To 1 Step -1
```

```
      If Mid$(sFilePath, i, 1) = "." Then
        iPoint = i
        Exit For
      End If
    Next i
    If iPoint = 0 Then iPoint = Len(sFilePath) + 1
    For i = iPoint - 1 To 1 Step -1
      If Mid$(sFilePath, i, 1) = "\" Then
        iBackSlash = i
        Exit For
      End If
    Next i
    sFileName = Mid$(sFilePath, iBackSlash + 1, iPoint - iBackSlash - 1)
    MsgBox sFileName
  End Sub
```

The first `For...Next` loop uses the `Len` function to determine how many characters are in the `sFilePath` variable, and `i` is set up to step backwards, counting from the last character position, working toward the first character position. The `Mid$` function extracts the character from `sFilePath` at the position defined by `i` and tests it to see if it is a period.

When a period is found, the position is recorded in `iPoint` and the first `For...Next` loop is exited. If the filename has no extension, no period is found and `iPoint` will have its default value of 0. In this case, the `If` test records an imaginary period position in `iPoint` that is one character beyond the end of the filename.

The same technique is used in the second `For...Next` loop as the first, starting one character before the period, to find the position of the backslash character, and storing the position in `iBackSlash`. The `Mid$` function is then used to extract the characters between the backslash and the period.

For Each...Next Loop

When you want to process every member of a collection, you can use the `For Each...Next` loop. The following example is a rework of the `FilePathInFooter` procedure:

```
Sub FilePathInFooter()
  Dim sFilePath As String, wks As Worksheet

  sFilePath = ActiveWorkbook.FullName
  For Each wks In Worksheets
    wks.PageSetup.CenterFooter = sFilePath
  Next wks
End Sub
```

The loop steps through all the members of the collection. During each pass, a reference to the next member of the collection is assigned to the object variable `wks`.

The following example lists all the files in the root directory of the C: drive. It uses the Windows Scripting `FileSystemObject` to create a reference to the C drive root directory. The example uses a `For Each...Next` loop to display the names of all the files in the directory:

```
Sub FileList()
  'Listing files with a For...Each loop

  Dim objFSO As Object
  Dim objFolder As Object
  Dim objFile As Object

  'Create a reference to the FileSystemObject
  Set objFSO = CreateObject("Scripting.FileSystemObject")

  'Create a folder reference
  Set objFolder = objFSO.GetFolder("C:\")

  'List files in folder
  For Each objFile In objFolder.Files
      MsgBox objFile.Name
  Next objFile

End Sub
```

The code uses techniques that are discussed in Chapter 19 to reference objects outside the Excel object model. If you test this procedure on a directory with lots of files, and get tired of clicking OK, don't forget that you can break out of the code with Ctrl+Break.

Arrays

Arrays are VBA variables that can hold more than one item of data. An array is declared by including parentheses after the array name. An integer is placed within the parentheses, defining the number of elements in the array:

```
Dim avData(2)
```

You assign values to the elements of the array by indicating the element number as follows:

```
avData(0) = 1
avData(1) = 10
avData(2) = 100
```

The number of elements in the array depends on the array base. The default base is 0, which means that the first data element is item 0. `Dim avData(2)` declares a three-element array if the base is 0. Alternatively, you can place the following statement in the declarations section at the top of your module to declare that arrays are 1-based:

```
Option Base 1
```

With a base of 1, `Dim avData(2)` declares a two-element array. Item 0 does not exist.

You can use the following procedure to test the effect of the `Option Base` statement:

```
Sub Array1()
  Dim aiData(10) As Integer
  Dim sMessage As String, i As Integer

  For i = LBound(aiData) To UBound(aiData)
    aiData(i) = i
  Next i
  sMessage = "Lower Bound = " & LBound(aiData) & vbCr
  sMessage = sMessage & "Upper Bound = " & UBound(aiData) & vbCr
  sMessage = sMessage & "Num Elements = " & WorksheetFunction.Count(aiData) & vbCr
  sMessage = sMessage & "Sum Elements = " & WorksheetFunction.Sum(aiData)
  MsgBox sMessage
End Sub
```

`Array1` uses the `LBound` (lower bound) and `UBound` (upper bound) functions to determine the lowest and highest index values for the array. It uses the `Count` worksheet function to determine the number of elements in the array. If you run this code with `Options Base 0`, or no `Options Base` statement, in the declarations section of the module, it will show a lowest index number of 0 and 11 elements in the array. With `Options Base 1`, it shows a lowest index number of 1 and 10 elements in the array.

> *Note the use of the intrinsic constant* `vbCr`*, which contains a carriage return character.* `vbCr` *is used to break the message text to a new line.*

If you want to make your array size independent of the `Option Base` statement, you can explicitly declare the lower bound as well as the upper bound as follows:

```
Dim avData(1 To 2)
```

Arrays are very useful for processing lists or tables of items. If you want to create a short list, you can use the `Array` function as follows:

```
Dim avData As Variant
avData = Array("North", "South", "East", "West")
```

You can then use the list in a `For...Next` loop. For example, you could open and process a series of workbooks called `North.xls`, `South.xls`, `East.xls`, and `West.xls`:

```
Sub Array2()
  Dim avData As Variant, wkb As Workbook
  Dim i As Integer

  avData = Array("North", "South", "East", "West")
  For i = LBound(avData) To UBound(avData)
    Set wkb = Workbooks.Open(Filename:=avData(i) & ".xls")
    'Process data here
    wkb.Close SaveChanges:=True
  Next i
End Sub
```

Multi-Dimensional Arrays

So far you have only looked at arrays with a single dimension. You can actually define arrays with up to 60 dimensions, although few people would use more than two or three dimensions. The following statements declare two-dimensional arrays:

```
Dim avData(10,20)
Dim avData(1 To 10,1 to 20)
```

You can think of a two-dimensional array as a table of data. The preceding example defines a table with 10 rows and 20 columns.

Arrays are very useful in Excel for processing the data in worksheet ranges. It can be far more efficient to load the values in a range into an array, process the data, and write it back to the worksheet, than to access each cell individually.

The following procedure shows how you can assign the values in a range to a `Variant`. The code uses the `LBound` and `UBound` functions to find the number of dimensions in `avData`. Note that there is a second parameter in `LBound` and `UBound` to indicate which index you are referring to. If you leave this parameter out, the functions refer to the first index:

```
Sub Array3()
  Dim avData As Variant, vUBound As Variant
  Dim Message As String, i As Integer

  avData = Range("A1:A20").Value
  i = 1
  Do
    Message = "Lower Bound = " & LBound(avData, i) & vbCr
    Message = Message & "Upper Bound = " & UBound(avData, i) & vbCr
    MsgBox Message, , "Index Number = " & i
    i = i + 1
    On Error Resume Next
    vUBound = UBound(avData, i)
    If Err.Number <> 0 Then Exit Do
    On Error GoTo 0
  Loop
  Message = "Number of Non Blank Elements =" _
                & WorksheetFunction.CountA(avData) & vbCr
  MsgBox Message
End Sub
```

The first time around, the `Do...Loop`, `Array3` determines the upper and lower bounds of the first dimension of `avData`, as `i` has a value of 1. It then increases the value of `i` to look for the next dimension. It exits the loop when an error occurs, indicating that no more dimensions exist.

By substituting different ranges into `Array3`, you can determine that the array created by assigning a range of values to a `Variant` is two-dimensional, even if there is only one row or one column in the range. You can also determine that the lower bound of each index is 1, regardless of the `Option Base` setting in the declarations section.

Dynamic Arrays

When writing your code, it is sometimes not possible to determine the size of the array that will be required. For example, you might want to load the names of all the `.xls` files in the current directory into an array. You won't know in advance how many files there will be. One alternative is to declare an array that is big enough to hold the largest possible amount of data — but this would be inefficient. Instead, you can define a dynamic array and set its size when the procedure runs.

You declare a dynamic array by leaving out the dimensions:

```
Dim avData()
```

You can declare the required size at run time with a `ReDim` statement, which can use variables to define the bounds of the indexes:

```
ReDim avData(iRows, iColumns)
ReDim avData(iminRow to imaxRow, iminCol to imaxCol)
```

`ReDim` will re-initialize the array and destroy any data in it, unless you use the `Preserve` keyword. `Preserve` is used in the following procedure that uses a `Do...Loop` to load the names of files into the dynamic array called `asFNames`, increasing the upper bound of its index by one each time to accommodate the new name.

The `Dir` function returns the first filename found that matches the wildcard specification in `sFType`. Subsequent usage of `Dir`, with no parameter, repeats the same specification, getting the next file that matches, until it runs out of files and returns a zero-length string:

```
Sub FileNames()
  Dim sFName As String
  Dim asFNames() As String
  Dim sFType As String
  Dim i As Integer

  sFType = "*.xls"
  sFName = Dir(sFType)
  Do Until sFName = ""
    i = i + 1
    ReDim Preserve asFNames(1 To i)
    asFNames(i) = sFName
    sFName = Dir
  Loop
  If i = 0 Then
    MsgBox "No files found"
  Else
    For i = 1 To UBound(asFNames)
      MsgBox asFNames(i)
    Next i
  End If
End Sub
```

If you intend to work on the files in a directory and save the results, it is a good idea to get all the filenames first, as in the `FileNames` *procedure, and use that list to process the files. It is not a good idea to rely on the* `Dir` *function to give you an accurate file list while you are in the process of reading and overwriting files.*

Run-Time Error-Handling

When you are designing an application, you should try to anticipate any problems that could occur when the application is used in the real world. You can remove all the bugs in your code and have flawless logic that works with all permutations of conditions, but a simple operational problem could still bring your code crashing down with a less than helpful message displayed to the user.

For example, if you try to save a workbook file to the floppy disk in the A: drive, and there is no disk in the A: drive, your code will grind to a halt and display a message that will probably not mean anything to the average user.

If you anticipate this particular problem, you can set up your code to gracefully deal with the situation. VBA allows you to trap error conditions using the following statement:

```
On Error GoTo LineLabel
```

`LineLabel` is a marker that you insert at the end of your normal code, as shown in the following code with the line label `errTrap`. Note that a colon follows the line label. The line label marks the start of your error recovery code and should be preceded by an `Exit` statement to prevent execution of the error recovery code when no error occurs:

```
Sub ErrorTrap1()
  Dim lAnswer As Long, sMyFile As String
  Dim sMessage As String, sCurrentPath As String

  On Error GoTo errTrap
  sCurrentPath = CurDir$

  ChDrive "A"
  ChDrive sCurrentPath
  ChDir sCurrentPath
  sMyFile = "A:\Data.xls"
  Application.DisplayAlerts = False
  ActiveWorkbook.SaveAs Filename:=sMyFile
TidyUp:
  ChDrive sCurrentPath
  ChDir sCurrentPath
Exit Sub
errTrap:
  sMessage = "Error No: = " & Err.Number & vbCr
  sMessage = sMessage & Err.Description & vbCr & vbCr
  sMessage = sMessage & "Please place a disk in the A: drive" & vbCr
  sMessage = sMessage & "and press OK" & vbCr & vbCr
  sMessage = sMessage & "Or press Cancel to abort File Save"
  lAnswer = MsgBox(sMessage, vbQuestion + vbOKCancel, "Error")
  If lAnswer = vbCancel Then Resume TidyUp
  Resume
End Sub
```

Once the `On Error` statement is executed, error trapping is enabled. If an error occurs, no message is displayed and the code following the line label is executed. You can use the `Err` object to obtain information about the error. The `Number` property of the `Err` object returns the error number, and the `Description` property returns the error message associated with the error. You can use `Err.Number` to

determine the error when it is possible that any of a number of errors could occur. You can incorporate `Err.Description` into your own error message, if appropriate.

> *In Excel 5 and 95,* `Err` *was not an object, but a function that returned the error number. Because* `Number` *is the default property of the* `Err` *object, using* `Err` *by itself is equivalent to using* `Err.Number`*, and the code from the older versions of Excel still works in Excel 97 and later versions.*

The code in `ErrorTrap1`, after executing the `On Error` statement, saves the current directory drive and path into the variable `sCurrentPath`. It then executes the `ChDrive` statement to try to activate the A: drive. If there is no disk in the A: drive, error 68 — (Device unavailable) occurs and the error recovery code executes. For illustration purposes, the error number and description are displayed and the user is given the opportunity to either place a disk in the A: drive and continue, or abort the save.

If the user wishes to stop, you branch back to `TidyUp` and restore the original drive and directory settings. Otherwise the `Resume` statement is executed. This means that execution returns to the statement that caused the error. If there is still no disk in the A: drive, the error recovery code is executed again. Otherwise the code continues normally.

The only reason for the `ChDrive "A"` statement is to test the readiness of the A: drive, so the code restores the stored drive and directory path. The code sets the `DisplayAlerts` property of the `Application` object to `False`, before saving the active workbook. This prevents a warning if an old file called `Data.xls` is being replaced by the new `Data.xls`. (See Chapter 3 for more on `DisplayAlerts`.)

The `Resume` statement comes in three forms:

- `Resume` causes execution of the statement that caused the error.
- `Resume Next` returns execution to the statement following the statement that caused the error, so the problem statement is skipped.
- `Resume LineLabel` jumps back to any designated line label in the code, so you can decide to resume where you want.

The following code uses `Resume Next` to skip the `Kill` statement, if necessary. The charmingly named `Kill` statement removes a file from disk. The following code removes any file with the same name as the one you are about to save, so there will be no need to answer the warning message about overwriting the existing file.

The problem is that `Kill` will cause a fatal error if the file does not exist. If `Kill` does cause a problem, the error recovery code executes and you use `Resume Next` to skip `Kill` and continue with `SaveAs`. The `MsgBox` is there for educational purposes only. You would not normally include it:

```
Sub ErrorTrap2()
  Dim sMyFile As String, sMessage As String
  Dim sAnswer As String

  On Error GoTo errTrap

  Workbooks.Add
  sMyFile = "C:\Data.xls"
  Kill sMyFile
```

```
    ActiveWorkbook.SaveAs Filename:=sMyFile
    ActiveWorkbook.Close

    Exit Sub
errTrap:
    sMessage = "Error No: = " & Err.Number & vbCr
    sMessage = sMessage & Err.Description & vbCr & vbCr
    sMessage = sMessage & "File does not exist"
    sAnswer = MsgBox(sMessage, vbInformation, "Error")
    Resume Next
End Sub
```

On Error Resume Next

As an alternative to `On Error GoTo`, you can use:

```
On Error Resume Next
```

This statement causes errors to be ignored, so it should be used with caution. However, it has many uses. The following code is a rework of `ErrorTrap2`:

```
Sub ErrorTrap3()
    Dim sMyFile As String

    Workbooks.Add
    sMyFile = "C:\Data.xls"
    On Error Resume Next
    Kill sMyFile
    On Error GoTo 0
    ActiveWorkbook.SaveAs Filename:=sMyFile
    ActiveWorkbook.Close
End Sub
```

Use `On Error Resume Next` just before the `Kill` statement. If `C:\Data.xls` does not exist, the error caused by `Kill` is ignored and execution continues on the next line. After all, you don't care if the file does not exist. That's the situation you are trying to achieve.

`On Error GoTo 0` is used to turn on normal VBA error-handling again. Otherwise, any further errors would be ignored. It is best not to try to interpret this statement, which appears to be directing error-handling to line 0. Just accept that it works.

You can use `On Error Resume Next` to write code that would otherwise be less efficient. The following sub procedure determines whether a name exists in the active workbook:

```
Sub TestForName()
    If bNameExists("SalesData") Then
        MsgBox "Name Exists"
    Else
        MsgBox "Name does not exist"
    End If
End Sub

Function bNameExists(sMyName As String) As Boolean
```

```
    Dim sName As String
    On Error Resume Next
    sName = Names(sMyName).RefersTo
    If Err.Number <> 0 Then
      bNameExists = False
      Err.Clear
    Else
      bNameExists = True
    End If
  End Function
```

`TestForName` calls the `bNameExists` function, which uses `On Error Resume Next` to prevent a fatal error when it tries to assign the name's `RefersTo` property to a variable. There is no need for `On Error GoTo 0` here, because error-handling in a procedure is disabled when a procedure exits, although `Err.Number` is not cleared.

If no error occurred, the `Number` property of the `Err` object is 0. If `Err.Number` has a non-0 value, an error occurred, presumably because the name did not exist, so `bNameExists` is assigned a value of `False` and the error is cleared. The alternative to this single pass procedure is to loop through all the names in the workbook, looking for a match. If there are lots of names, this can be a slow process.

Summary

In this chapter, you have seen those elements of the VBA language that enable you to write useful and efficient procedures. You have seen how to add interaction to macros with the `MsgBox` and `InputBox` functions, how to use variables to store information, and how to get help with VBA keywords.

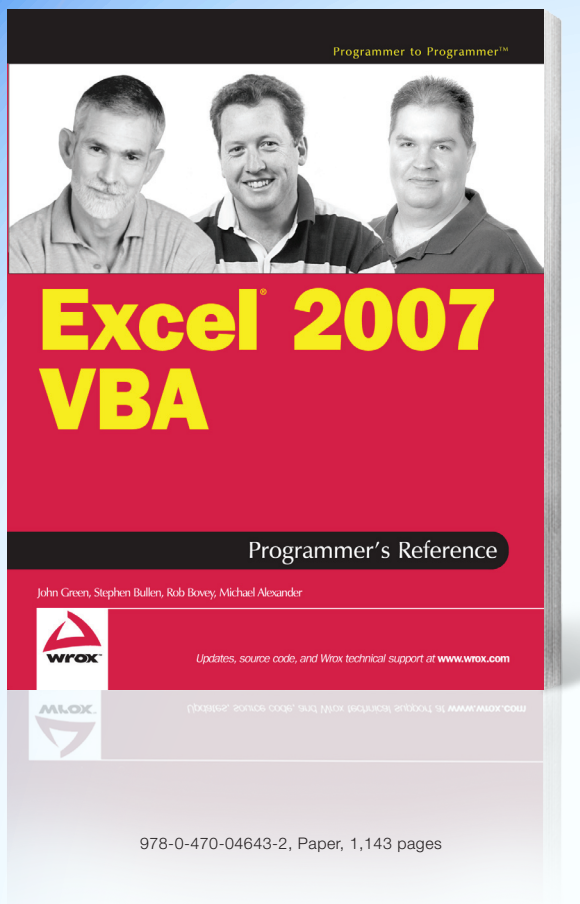
You have seen how to declare variables and define their type, and the effect on variable scope and lifetime of different declaration techniques. In addition, you used the block `If` and `Select Case` structures to perform tests and carry out alternative calculations, and `Do...Loop` and `For...Next` loops that allow you to efficiently repeat similar calculations. You have seen how arrays can be used, particularly with looping procedures. Moreover, you learned how to use `On Error` statements to trap errors.

When writing VBA code for Excel, the easiest way to get started is to use the macro recorder. You can then modify that code, using the VBE, to better suit your purposes and to operate efficiently. Using the Object Browser, Help screens, and the reference section of this book, you can discover objects, methods, properties, and events that can't be found with the macro recorder. Using the coding structures provided by VBA, you can efficiently handle large amounts of data and automate tedious processes.

You now have the knowledge required to move on to the next chapter, where you will find a rich set of practical examples showing you how to work with key Excel objects. You will discover how to create your own user interface, setting up your own Ribbon buttons and dialog boxes, and embedding controls in your worksheets to enable yourself and others to work more productively.

Ready for more?

CHECK OUT THIS BOOK IN ITS ENTIRETY.



978-0-470-04643-2, Paper, 1,143 pages

Visit us at wrox.com

Get ready to take your Excel applications to the next level by harnessing the power of the VBA language. This comprehensive resource will help you gain more control over your spreadsheets by using VBA while also showing you how to develop more dynamic Excel applications for other users. From introductory concepts to advanced developer topics, it guides you through every aspect of Excel 2007, including the Ribbon and the XML file formats.

In order to master all of the new features of this program, you'll find an introduction to VBA and details on how to use it to enhance Excel. You'll then learn how to work with the key objects and uncover the best ways to gain access to workbooks, worksheets, charts, and more. And you'll find out how to write code for international compatibility, program the Visual Basic Editor, and use the functions in the Win32 API, which will expand your Excel VBA programming skills.

What you will learn from this book

- How to write code that is readable, easy to maintain, and runs at maximum speed
- Tips for utilizing the Visual Basic® Editor and its multitude of tools
- Techniques for accessing data in a range of formats
- Ways to set up your applications and convert them to add-ins
- How to manipulate the Office XML file formats
- A thorough explanation of RibbonX
- Best practices for managing external data and using OLAP data sources
- Methods for effectively debugging your application
- Tips for packaging and distributing customized applications to other users

Who is this book for?

This book is for Excel users and programmers from beginning to advanced. You should have a reasonable working knowledge of Excel and a full installation of the software.

Wrox Programmer's References are designed to give the experienced developer straight facts on a new technology, without hype or unnecessary explanations. They deliver hard information with plenty of practical examples to help you apply new tools to your development projects today.

Contents:

wrox™
An Imprint of WILEY