Visual Basic® User's Guide

Microsoft® Office

utomating, Customizing, and Programming in Microsoft Excel
ith the Microsoft Visual Basic Programming System,
pplications Edition

Microsoft® EXCEL

Visual Basic® User's Guide

Microsoft® Excel

Version 5.0

Automating, Customizing, and Programming in Microsoft Excel with the Microsoft Visual Basic Programming System, Applications Edition

Microsoft Corporation

The Microsoft Excel Analysis ToolPak was developed by GreyMatter International, Inc., 173 Otis Street, P.O. Box 388, Cambridge, MA 02141.

The Microsoft Excel Solver program was developed by Frontline Systems, Inc., P.O. Box 4288, Incline Village, Nevada 89450-4288. Portions of the Microsoft Excel Solver program code are copyright 1990, 1991, and 1992 by Frontline Systems, Inc. Portions are copyright 1989 by Optimal Methods, Inc.

The Microsoft Excel Solver uses GRG2 nonlinear optimization code developed by Leon Lasdon, University of Texas at Austin, and Allan Waren, Cleveland State University. Linear and integer problems use the simplex method with bounds on the variables and the branch and bound method, implemented by John Watson and Dan Fylstra, Frontline Systems, Inc.

Microsoft Word was used in the production of this manual.

Document No. XL57927-0694

Printed in Ireland 10

Contents

Document Conventions

This manual uses the typographic conventions shown in the following table. You might not recognize all the terms or Visual Basic® keywords, but you'll learn more about them later.

Example of convention	Description
setup	Words or characters you're instructed to type appear in bold.
object	In text, italic type indicates important new terms, usually the first time they occur in the book.
propertyname	In code syntax, italic type indicates placeholders for information you supply.
EXCEL.EXE (Microsoft® Windows™) MICROSOFT EXCEL (Apple Macintosh)	Words in all capital letters indicate filenames. All capital letters are also used for directory and folder names.
ENTER	Small capital letters are used for the names of keys and key combinations, such as ENTER and CTRL+R.
CTRL+V	A plus sign (+) between key names indicates a combination of keys. For example, CTRL+V means to hold down the CTRL key while pressing the V key.
DOWN ARROW	Individual arrow keys are referred to by the direction of the arrow on the key (LEFT, RIGHT, UP, or DOWN). The phrase "arrow keys" is used when describing these keys collectively.
BACKSPACE, HOME	Other navigational keys are referred to by their specific names.
[*expressionlist*]	In syntax, items inside square brackets are optional.
{**While** | **Until**}	In syntax, a vertical bar indicates a choice between two or more items. You must choose one of the items. Braces are sometimes used to group items separated by a vertical bar.
`Syntax error`	This font is used for user-defined variables and error messages.

Example of convention	Description

`Sub StockSale ( )`
`.`
`.`
`.`
`End Sub`
	A column or row of three periods tells you that part of an example program has been intentionally omitted.

Sub, **If**, **ChDir**, **MsgBox**, **True**	Words in bold with the initial letter capitalized indicate a Visual Basic keyword—a language-specific term.
Add, **Height**, Application, Range, Row	Names of properties, methods, and objects appear with initial letter capitalized. **Height** is also bold because it is a property, and so is **Add** because it is a method..

This manual also uses the general conventions and mouse and keyboard conventions that are used throughout the *Microsoft Excel User's Guide.*

Programming Style in This Manual

The following guidelines are used in writing programming code in this manual. For more information about Visual Basic code, see Chapter 6, "Working with Visual Basic Code in Procedures."

- This font is used for code:

```
Sub HelloWorld
Cells(1,1).Value = "Hello, world!"
End Sub
```

- An apostrophe (') introduces comments in code.

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Names of macros, user-defined functions, arguments, and variables appear with initial letters capitalized throughout this manual. Note that macro and function names cannot include spaces, so if the name consists of more than one word, the other words in the name also have their initial letters capitalized.

```
' The AuditResult user-defined function is in the Finance module.
Function AuditResult (LatestIncome, Expenses, Taxes, Commissions)
```

Many macro developers prefer to use the underscore character (_) to separate individual words in the names of macros and functions (procedures). Also, some developers prefer variables and arguments with an initial lowercase letter, especially when variables may be confused with property names.

```
' The Stock_Expense macro is in the Finance module.
Sub Stock_Expense (sharesSold, pricePerShare)

soundRecordCapable = Application.CanRecordSounds
```

- Keywords appear with initial letters capitalized, while built-in constants appear with an initial lowercase "xl."

```
' Sub is a keyword.
Sub Title (TitleText)

' xlManual is a built-in constant.
Application.Calculation = xlManual
```

- Control-flow blocks and statements in **Sub** and **Function** procedures are indented within the code that surrounds them.

```
Sub CheckRecordSound
    SoundRecordCapable = Application.CanRecordSounds
    If SoundRecordCapable Then
        Cells(1,1).SoundNote.Record
    End If
End Sub
```

- The line-continuation character, an underscore (_), indicates that code continued from one line to the next in the book is part of the same logical line. You can type these statements all on one line in the Visual Basic module. You can also divide lines of code and add the line-continuation character yourself.

```
ActiveSheet.Rectangles.Add _
    width:=200, _
    height:=200, _
    left:=50, _
    top:=50
```

Note The decimal separator, list separator, currency symbol, and date format that you use when you're writing or editing code are determined by the Language/Country settings on the Module General tab of the Options dialog box (Tools menu). The decimal separator, list separator, currency symbol, and date formats that users see when they run your code are determined by the country settings of their operating system.

The code examples in this book assume you use U.S. country settings. For more information about writing applications for use in other countries, see Appendix A, "Writing Code for International Use."

A Note About Terminology

The Introduction and Chapters 1 through 3 in this manual use the words *macro* and *user-defined function* to describe the two primary automation and customization tools in Microsoft Excel. Although experienced users of Microsoft Excel and other spreadsheet programs are familiar with these terms, they are not typically used in Visual Basic.

For this reason, in Chapter 4, "Introducing Visual Basic Procedures," the terms macro and user-defined function are replaced with a standard Visual Basic term—*procedure*. The meaning is nearly synonymous; macros and user-defined functions are each a type of procedure. All these terms are fully explained in Chapter 4.

Introduction

Welcome to Visual Basic—the powerful and easy-to-use programming language used in Microsoft Excel. Visual Basic refers to the Microsoft Excel implementation of the Microsoft Visual Basic Programming System, Applications Edition. With Visual Basic, you can automate everyday tasks, add custom features and functions to suit your needs, and even create complete applications.

In Microsoft Excel, you automate tasks with macros. A *macro* is a sequence of instructions that tells Microsoft Excel what to do. The instructions are written in Visual Basic—a computer programming language—but you don't have to be a programmer to create macros; in fact, you don't even have to learn Visual Basic to put macros to work.

Microsoft Excel provides a Macro Recorder that writes macros for you. The Macro Recorder stores the actions you take and the commands you choose as you work with Microsoft Excel. Later you can play back, or run, the macro to repeat your recorded actions automatically, saving you time and effort. You'll learn about the Macro Recorder in Chapter 1, "Automating Repeated Tasks."

After you learn to record and run macros, you'll discover how useful they can be, and you'll probably want to make them even more powerful by adding your own Visual Basic code. This manual shows you how to do that, and Visual Basic is easy to learn. Whether you're a new user or an experienced macro developer, Visual Basic can make you more productive.

With Visual Basic, you can create custom commands, menus, dialog boxes, messages, and buttons, and you can display custom online Help for all of these items. You can transform Microsoft Excel into an entirely different application. From automating repetitious tasks to developing powerful, full-featured applications, Visual Basic contains the tools to make Microsoft Excel fit your specific needs.

Getting Started

The rest of this chapter describes the printed and online documentation that you will use as you learn more about Visual Basic. This manual assumes you already know how to use Microsoft Excel. If you're new to Microsoft Excel, you should set this manual aside for now. For general information about Microsoft Excel, see the *Microsoft Excel User's Guide.*

Where to Find Information About Visual Basic

Your Microsoft Excel package includes the following four sources of information about Visual Basic:

- The *Microsoft Excel Visual Basic User's Guide* (this manual) teaches you how to record, run, and edit macros; how to create and enter user-defined functions; and how to program in Visual Basic so that you can write your own macros or even create complete applications.
- The main Microsoft Excel Help file provides general online Help about macros (what they are and how to record and run them).
- Examples and Demos are lessons within online Microsoft Excel Help that show you how to perform common macro tasks.

  For more information about Help and how to use it, see Chapter 5, "Getting Information While You Work," in the *Microsoft Excel User's Guide.*
- The Visual Basic Reference is a detailed online reference for Visual Basic functions, statements, methods, properties, and objects.

Note The *Microsoft Excel Visual Basic Reference* is also available in printed form and can be purchased in book and software stores or ordered directly from Microsoft Press. You can also order *Microsoft Excel Visual Basic Step by Step,* a self-paced tutorial with timesaving examples on disk that makes learning Visual Basic in Microsoft Excel easier. To place a credit card order, call 615-793-5090, or call toll-free 800-MS-PRESS. Please be sure to have your reference code FXL ready for faster order processing. CompuServe members can also order through the Microsoft Press Electronic Book Store by typing **GO MSP**.

How This Book Is Organized

The chapters and appendixes of the *Microsoft Excel Visual Basic User's Guide* are arranged in order of increasing complexity. Some users might want to run and record macros and create user-defined functions but not learn Visual Basic. Others might want to learn part of Visual Basic but do not plan to create custom applications. Still others might want to exploit the advanced features. The chapters are organized to address the needs of all these users.

- Chapters 1 through 3 describe macros and explain how to record, run, display, and edit them, and how to create and enter user-defined functions. These chapters are written for all users who are interested in automating their work, but particularly for those who are not experienced macro developers.
- Chapter 4 introduces Visual Basic and replaces the term macro with a new term used in Visual Basic — procedure. This chapter also explains how to manage multiple procedures for more complex tasks.
- Chapters 5 through 10 describe the Visual Basic language in more detail. They show you how to use code in procedures, pass arguments to and return values from procedures, control the way your code runs, work with objects and their properties, debug your code, and work with other applications. These chapters are written for users who want to go beyond automating simple tasks.
- Chapters 11 through 13 describe how to customize Microsoft Excel to suit your needs and how to create a custom application. Specific topics include creating custom dialog boxes, commands, menus, buttons, and add-in applications. These chapters are written for experienced macro writers, but users who simply want to customize menus and toolbars will find useful information here, too.
- Appendixes A through D explain how to write international applications, provide helpful information for experienced Microsoft Excel users who are switching to Visual Basic, describe creating custom Help files for Visual Basic applications, and list all the toolbar buttons in Microsoft Excel version 5.0.

Code Samples

Your Microsoft Excel package includes code samples that you can open and run. Not only are these code samples excellent learning tools, but they are also useful. You can copy any part of them into your own applications, modifying them as necessary. These code samples are located in the EXAMPLES directory or folder of the directory or folder in which you installed Microsoft Excel.

Using Online Help

Microsoft Excel provides extensive online Help for the Visual Basic language, the objects Microsoft Excel supports, and the properties and methods of those objects.

You can access the Visual Basic Reference in one of the following four ways:

- In a Visual Basic module, select a property, statement, or other keyword, and then (in Microsoft Excel for Windows) press F1 to get context-sensitive Help. In Microsoft Excel for the Macintosh, press COMMAND+SLASH.
- In Microsoft Excel for Windows, choose the Contents command from the Help menu; in Microsoft Excel for the Macintosh, choose Microsoft Excel Help from the Help menu (which is the ? menu) or the Window menu. Click the "Programming with Visual Basic" topic.

- In the Help window, choose the Search button, and then search for a specific topic or Visual Basic term.
- From the View menu, choose the Object Browser command, and then choose the Element Help button (appears as a question mark) for information about an object, method, property, or function.

Setting Up the Visual Basic Reference

When you install Microsoft Excel, the Visual Basic Reference is installed automatically unless you specify that it shouldn't be. (The filename is VBA_XL.HLP in Microsoft Windows or MS EXCEL VISUAL BASIC HELP on the Apple Macintosh.) If you do not plan to use Visual Basic, you do not have to install the Visual Basic Reference, and you can skip the rest of this section and proceed to Chapter 1. Limited information about running and recording macros is in the main Help file (MAINXL.HLP in Windows or MS EXCEL HELP on the Macintosh).

If you plan to do anything more than record macros and if you have not already installed the Visual Basic Reference, you should run the Microsoft Excel Setup program and install it now. The information in the Visual Basic Reference will be extremely helpful to you as you develop your applications. For more information about the Setup program and installing optional files, see Chapter 1, "Installing and Starting Microsoft Excel," in the *Microsoft Excel User's Guide*.

Using the Visual Basic Reference

The Visual Basic Reference provides information about all the keywords in the Visual Basic language as well as some procedural information and miscellaneous reference information not related to keywords. To see how this information is organized, you can display the Contents screen of the Visual Basic Reference.

▶ **To display the Contents screen of the Visual Basic Reference**

1. From the Help menu, choose Contents.
2. Click the "Programming with Visual Basic" topic.

The Contents screen of the Visual Basic Reference directs you to information about the following:

- Common Visual Basic tasks and procedures
- Visual Basic functions
- Microsoft Excel objects in Visual Basic
- Properties associated with objects
- Methods associated with objects
- Visual Basic statements

- Visual Basic keywords listed by programming task
- Miscellaneous reference information

Most of the information in the Visual Basic Reference is about keywords. A *keyword* is a word or symbol that is recognized as part of the Visual Basic programming language. Statements, functions, objects, properties, and methods are examples of keywords. Even if you do not yet know what these words mean or much about Visual Basic code, you'll find the Visual Basic Reference useful as you explore the Visual Basic language.

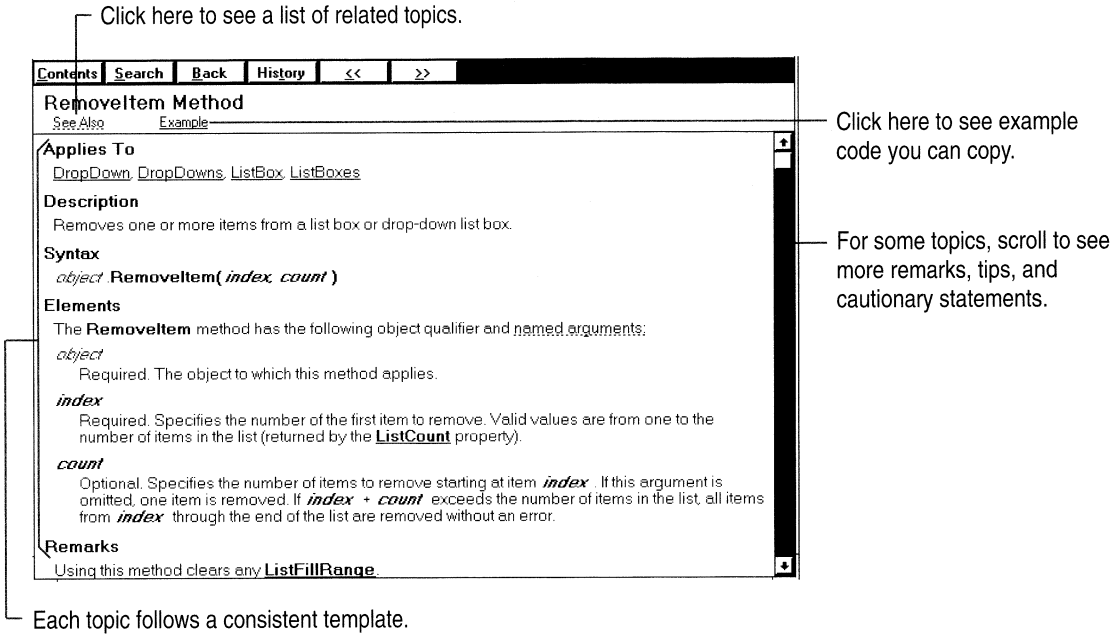
The Visual Basic Reference uses a template to describe its keywords. The items that make up the template are shown in the following table. The Help topic for a given keyword rarely contains all the items.

Heading in the Help topic	Information provided about the keyword
Title (not labeled)	The keyword whose topic you are reading.
See Also	A list of topics related to the keyword.
Example	Sample code containing the keyword. You can copy the code and paste it into your macro.
Applies To	A list of the objects to which the method or property applies. Appears only in descriptions of methods and properties.
Description	What the keyword is, does, or returns.
Syntax	The *syntax* of the keyword—the arguments to the keyword in the correct order.
Elements	A description of the elements in the syntax, including whether those elements are optional or required and what values they can take.
Remarks	Miscellaneous information about the keyword, its usage, or its syntax.

Within the Syntax and Elements sections, items that you must type (the keywords themselves, for example) are bold. Placeholders for the items that you must specify (objects, for example) are italic. Other placeholders (arguments, for example) are italic and bold. You'll learn more about arguments in Chapter 3, "Creating User-Defined Functions."

The Visual Basic Reference contains examples for nearly every Visual Basic keyword. You can click the Example topic to display sample code in another window, and then you can copy that code and paste it into your own code. Some of the examples are complete procedures; most are just one or two lines. You'll learn more about pasting sample code into your macros in Chapter 2, "Editing Recorded Macros."

The following illustration shows a typical Help topic in the Visual Basic Reference.



Information for Experienced Microsoft Excel Macro Developers

If you're an experienced Microsoft Excel developer, you have invested a great deal of time and effort in developing macros using the Microsoft Excel 4.0 macro language. Microsoft Excel version 5.0 fully supports the previous macro language in the following ways:

- All macros written for previous versions of Microsoft Excel will run in version 5.0.
- Microsoft has enhanced and updated the Microsoft Excel 4.0 macro language to support the new features of version 5.0.
- You can still create new macro sheets and record and write macros in the previous macro language.
- Online Help includes the Macro Functions Help, which is fully updated.

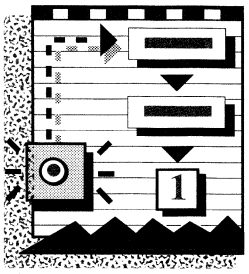
Microsoft will not update the Microsoft Excel 4.0 macro language for future versions of Microsoft Excel, so you are encouraged to write your new macros in Visual Basic. If you want older macros to support features that might be added to future versions of Microsoft Excel, you can rewrite those macros as Visual Basic

procedures, or you can call the old macros from new Visual Basic modules. Microsoft Excel does not translate macros written in the previous language into Visual Basic.

Your Microsoft Excel package provides information to help you make the transition from the Microsoft Excel 4.0 macro language to Visual Basic. For example, the code samples described in the previous section are provided in two versions, one in Visual Basic and one in the previous macro language. You can examine the two versions to see how the languages differ. For more information about making the transition to version 5.0, see Appendix B, "Switching from the Microsoft Excel 4.0 Macro Language."

CHAPTER 1

Automating Repeated Tasks



As you work in Microsoft Excel, you might find that you perform certain tasks routinely. For example, regularly updating sales figures, plotting data on a chart, or applying special formats can require you to repeat the same sequence of actions and commands. You can automate many of these tasks, saving you time and effort, by using the powerful Visual Basic programming language in Microsoft Excel.

You don't need to understand computer programming to begin to put Visual Basic to work. Microsoft Excel includes a *Macro Recorder*—a tool built into Microsoft Excel that creates Visual Basic code for you. This chapter describes the basic steps for using the Macro Recorder to automate simple tasks in Microsoft Excel. But recording macros is only the beginning. You usually need to edit and customize recorded macros to suit your needs. You can view and edit the Visual Basic code you've recorded in Chapter 2, "Editing Recorded Macros."

Contents

How Macros Simplify Tasks

Microsoft Excel automates tasks with macros. A macro is a series of commands that Microsoft Excel carries out automatically. For example, to format a range of cells you might choose the Cells command from the Format menu; select the Font tab; select a font name, style, and size; and then choose the OK button. With a macro, you can combine all these tasks and accomplish them in a single step.

By recording your own macros, you can tailor Microsoft Excel to your own needs and work even more efficiently. Any sequence of actions that you perform can be recorded. Then you can play back, or *run,* the macro to repeat your recorded actions automatically. Once you record a macro, you can also assign it to a menu item or a button. Then running the macro is as simple as choosing the menu item or clicking the button.

When to Record a Macro

You should consider recording a macro any time you find yourself regularly typing the same keystrokes, choosing the same commands, or going through the same sequence of actions. Some everyday tasks you can automate with macros include:

- Opening a group of workbooks and retrieving information from them.
- Printing several ranges of cells.
- Opening a database, sorting it, creating a report, and closing it.
- Setting up a new worksheet by entering titles, adjusting column widths, and applying special formats.

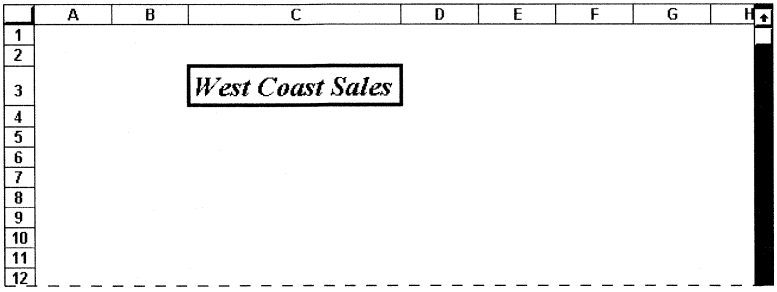
Example

Suppose you often need to set up a new worksheet in which you enter sales data. After switching to a new worksheet, you perform the following actions:

- Turn off the gridlines.
- Select cell C3.
- Enter the title **West Coast Sales**
- Format the title in the 18-point size of the Times New Roman font.
- Make the title bold and italic.
- Apply a dark blue border around the cell.
- Widen column C to accommodate the title.

To speed up the process of setting up this worksheet, you can record a macro to handle the entire task for you. Later, when you run the macro, Microsoft Excel automatically sets up your worksheet using the same sequence of steps you performed.

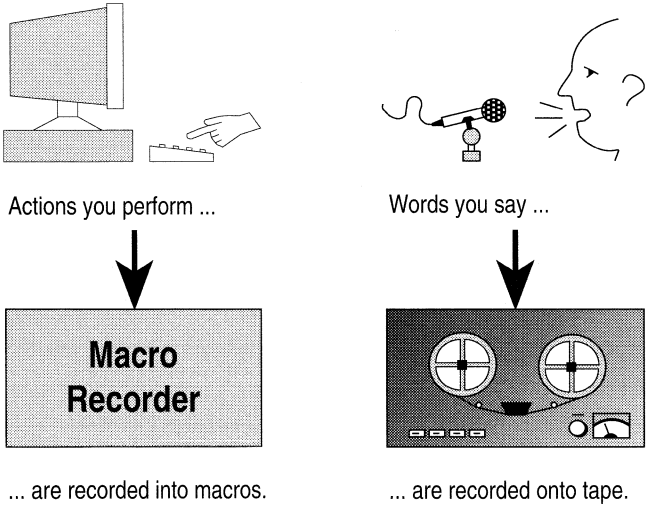
The completed worksheet is shown in the following illustration.



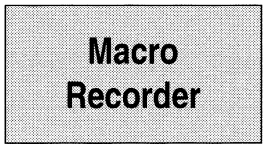
Note You can also use a template to set up worksheets with titles and formatting, but the benefit of using a macro is that you can later expand and customize it to automate even more complex tasks. For example, you can have the macro display a message box in which you type the title text or its location on the worksheet. You will learn how to customize and expand your macros throughout this manual.

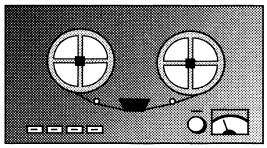
The Recording Process

In Microsoft Excel, the Macro Recorder stores actions you perform or commands you choose as you work. The Macro Recorder works similarly to a tape recorder. Just as a tape recorder records what you say, the Macro Recorder records what you do.

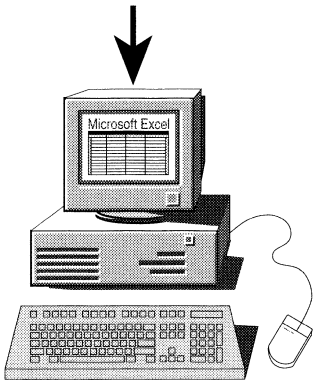


When you run the macro, the recorded sequence of instructions tells Microsoft Excel what to do. The Macro Recorder plays back what you did, just as a tape recorder plays back what you said.



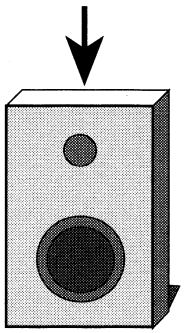


When you run a recorded macro ...



... Microsoft Excel performs actions automatically.

When you play back a tape ...



... recorded sound is emitted.

Recording a Macro

The first step in automating a task in Microsoft Excel is to record a macro. Following are the general steps to record a macro, and then a step-by-step example showing how the procedure works.

▶ **To record a macro**

1. From the Tools menu, choose Record Macro, and then choose Record New Macro.
2. In the Macro Name box, type a name for the macro.

   The name can contain letters, numbers, and underscores; it must begin with a letter. The name can't contain spaces or punctuation marks.
3. In the Description box, type a description of the macro.

4. To set options for the macro, choose the Options button, and then set any of the options.

   For a description of these options, choose the Help button.

5. Choose the OK button.

   While the Macro Recorder is working, the Stop Macro button appears on your screen on its own toolbar.

6. Carry out the actions that you want to record.

7. Click the Stop Macro button.

   You can also choose Record Macro from the Tools menu and then choose Stop Recording.

Tip To speed up the recording process, you can skip steps 2 and 3 in the preceding step-by-step procedure and let Microsoft Excel name your macro automatically. The macro receives the name "Macro*n*," where *n* is the first number that gives the macro a unique name.

Example

The following procedures show you how to record a macro that sets up a worksheet by turning off gridlines, entering a title into cell C3, and then formatting the cell with custom formats.

You begin by turning on the Macro Recorder and giving your macro a name and a description.

▶ **To begin recording the sample macro**

1. From the Tools menu, choose Record Macro, and then choose Record New Macro.
2. In the Macro Name box, type **CreateTitle**
3. In the Description box, type **Sets up a worksheet with a title in cell C3 and applies custom formats**
4. Choose the OK button.

Next, you turn off the gridlines on the worksheet.

▶ **To turn off gridlines**

1. From the Tools menu, choose Options.
2. Select the View tab.
3. Under Window Options, clear the Gridlines check box.
4. Choose the OK button.

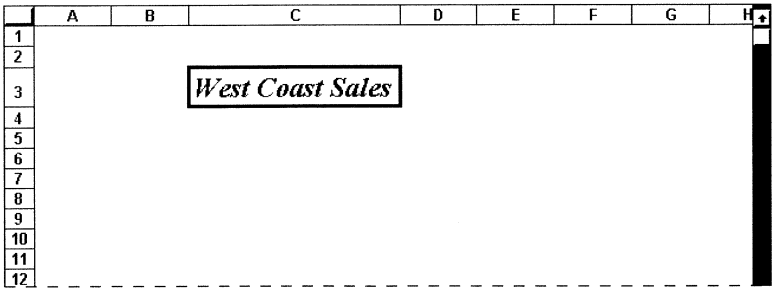
Finally, you enter a title in the cell, apply your custom formatting, and conclude the macro. This example assumes the Move Selection After Enter option on the Edit tab of the Options dialog box is turned off.

▶ **To enter the title, apply custom formatting, and conclude the macro**

1. Select cell C3 and type **West Coast Sales**, and then press ENTER.
2. From the Format menu, choose Cells.
3. Select the Font tab.
4. In the Font box, select Times New Roman.
5. In the Font Style box, select Bold Italic.
6. In the Size box, select 18.
7. Select the Border tab.
8. Under Border, select Outline.
9. Under Style, select the thickest solid line.
10. Choose the OK Button.
11. From the Format menu, choose Column, and then choose AutoFit Selection.
12. Click the Stop Macro button.

    You can also choose Record Macro from the Tools menu and then choose Stop Recording.

Your worksheet should look similar to the following illustration.



Running a Macro

After you record a macro, you can play back, or run, the macro at any time; Microsoft Excel carries out all the commands stored in the macro.

▶ **To run a macro**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select a name.
3. Choose the Run button.

Note You can interrupt a running macro by pressing the ESC key (or COMMAND+PERIOD on the Macintosh). This is helpful if you run the wrong macro or decide you don't need to finish running a macro. When you interrupt a macro, Microsoft Excel displays the Macro Error dialog box. To learn more about the options in this dialog box, choose the Help button. To close the dialog box, press the ESC key again or choose the End button.

Example

You can run the macro you recorded in the previous example.

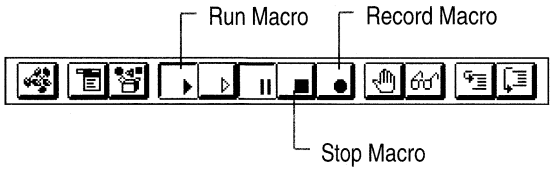
▶ **To run the CreateTitle macro**

1. Switch to a new worksheet in your workbook.
2. Select cell A1.
3. From the Tools menu, choose Macro.
4. In the Macro Name/Reference box, type or select **CreateTitle**
5. Choose the Run button.

Your worksheet should look similar to the illustration in the previous section.

Using the Visual Basic Toolbar

Microsoft Excel provides a Visual Basic toolbar that you can display by choosing the Toolbars command from the View menu. You can use several of the buttons on this toolbar to record, run, and stop recording macros.



The other buttons on the Visual Basic toolbar are described throughout this manual and in Help.

Making Your Macro Easier to Use

Once you record a macro, you can attach it to a menu item or a button or other graphic object, making it as accessible and convenient as built-in menu commands or buttons in Microsoft Excel. Not only does this speed up your work, it also makes it easier for other users to accomplish the same tasks simply by running your macro.

Adding a Macro to the Tools Menu

If you add a macro to the Tools menu, you can choose it the same way you choose any other menu command in Microsoft Excel.

▶ **To add a macro to the Tools menu**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select a macro name.
3. Choose the Options button.
4. Under Assign To, select the Menu Item On Tools Menu check box, and then type the command name as you want it to appear on the Tools menu.
5. Choose the OK button.
6. Choose the Close button.

Assigning a Macro to a Button on a Sheet

In Microsoft Excel, you can create a button on a worksheet or chart sheet and then assign a macro to it. By attaching a macro to a button, you make it visible and readily available while you are working. If the button appears on a worksheet, for example, the macro is available every time you open that worksheet.

▶ **To create a button on a sheet and assign a macro to it**

Before you do this procedure, you must have the Drawing toolbar displayed. Use the Toolbars command on the View menu to display the toolbar.

1. Click the Create Button button on the Drawing toolbar.
2. Point to where you want one corner of the button.
3. Drag until the button is the size and shape you want.

   When you release the mouse button, the Assign Macro dialog box appears.
4. To assign an existing macro to the button, type or select the name of the macro in the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To assign a new macro to the button, choose the Record button, and then follow the steps for recording a macro.

Note You can choose the Object command on the Format menu to change the font and alignment of the button text, or to change the button's protection, positioning, and printing properties.

You can also change the default name of a button or other graphic object by selecting the object and typing a new name in the Name/Reference area to the left of the formula bar. You'll use this name to refer to the object in Visual Basic code.

Assigning a Macro to a Button on a Toolbar

When you assign a macro to a button on a toolbar, your macro is available to every sheet in the workbook, if the toolbar is displayed. Typically, the toolbar button to which you assign your macro is an unused custom button. *Custom buttons* are buttons Microsoft Excel provides especially for macros or other customization needs; they can be found in the Custom button category, as demonstrated following.

You can also assign your macro to a built-in button that Microsoft Excel already uses. This overrides the button's normal function with that of your macro.

After you assign your macro to a built-in or custom button, you can place the button on any built-in toolbar, or you can create a new toolbar. After you place the button on a toolbar, it functions similarly to any built-in toolbar button, such as the Sum button or the Print button, except that it runs the macro that you assign to it.

▶ **To assign a macro to a built-in toolbar button**

1. From the View menu, choose Toolbars.
2. Choose the Customize button.

   The Customize dialog box appears, suspending the built-in toolbar button's normal function.
3. If the built-in button is currently displayed on a toolbar, click the button.

   –Or–

   If the built-in button is not displayed, select a category in the Categories box, and then drag the button from the box to any visible toolbar.
4. From the Tools menu, choose Assign Macro.
5. To assign an existing macro to the built-in button, type or select the name of the macro in the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To assign a new macro to the built-in button, choose the Record button, and then follow the steps for recording a macro.
6. If necessary, choose the Close button to close the Customize dialog box.

Instead of overriding the normal function of a built-in button, you might drag one of the unused custom buttons onto a toolbar and then assign a macro to it.

▶ **To assign a macro to a custom button and place the button on a toolbar**

1. From the View menu, choose Toolbars.
2. To add the custom button to a new toolbar, select the text in the Toolbar Name box, type the name of the new toolbar, and then choose the New button.

   –Or–

   To add the custom button to an existing toolbar, choose the Customize button.
3. In the Categories box, select Custom.
4. Drag the custom button from the box to any visible toolbar.

   The button is added wherever you position it on the toolbar, and the Assign Macro dialog box appears.
5. To assign an existing macro to the custom button, type or select the name of the macro in the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To assign a new macro to the custom button, choose the Record button, and then follow the steps for recording a macro.
6. If necessary, choose the Close button to close the Customize dialog box.

Example

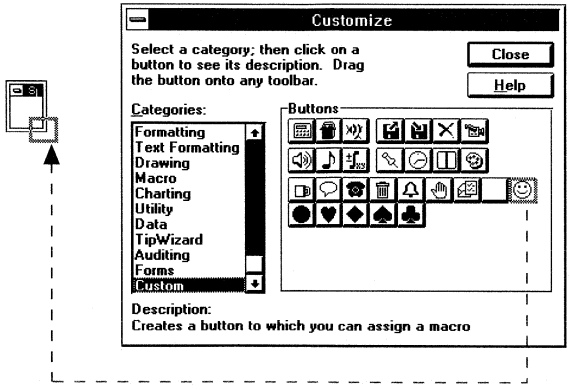
Here's how to assign the CreateTitle macro to a custom button and place the button on a new toolbar. To create the CreateTitle macro, follow the steps of the example in "Recording a Macro" earlier in this chapter.

▶ **To assign the CreateTitle macro to a custom button on a new toolbar**

1. From the View menu, choose Toolbars.
2. Select the text in the Toolbar Name box, and then type **Special Formatting**
3. Choose the New button.

   The new Special Formatting toolbar appears; it contains no toolbar buttons.
4. In the Categories box, select Custom.
5. Drag the blank toolbar button onto the Special Formatting toolbar, as shown in the following illustration.

   The Assign Macro dialog box appears.
6. In the Macro Name/Reference box, type or select **CreateTitle**
7. Choose the OK button.
8. Choose the Close button.



Assigning a Macro to a Graphic Object

Sometimes you might want to attach your macro to a graphic object. A *graphic object* is a picture that you paste into Microsoft Excel from another application or a graphic that you create using the buttons on the Drawing toolbar. A graphic object can make your macro easier to remember and use and make your worksheet more attractive. Assigning a macro to a graphic object is done in much the same way that you assign a macro to a button on a worksheet or chart sheet.

For example, you might record a macro that writes a monthly report on an international sales region. You can assign that macro to a flag representing the region. To create the report, you simply click the flag.

▶ **To assign a macro to a graphic object**

1. Click the graphic object to select it.

   If the object is a worksheet button or already has a macro assigned to it, hold down CTRL (in Microsoft Excel for Windows) or COMMAND (in Microsoft Excel for the Macintosh), and then click the object.

2. From the Tools menu, choose Assign Macro.

3. To assign an existing macro to the graphic object, type or select the name of the macro in the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To assign a new macro to the graphic object, choose the Record button, and then follow the steps for recording a macro.

For more information about creating graphic objects, see Chapter 13, "Creating Graphic Objects on Worksheets and Charts," in the *Microsoft Excel User's Guide.*

Changing the Macro Assigned to a Button or Graphic Object

You might find that you need to change the macro assigned to a button or graphic object, or record a new macro for an existing object, or assign an existing macro to a new object. Microsoft Excel makes these tasks easy to accomplish. The following procedure explains how to change the macro assigned to a graphic object or button on a worksheet or chart sheet.

▶ **To change the macro assigned to a button or graphic object on a sheet**

1. In Microsoft Excel for Windows, hold down CTRL, and then select the button or graphic object.

   In Microsoft Excel for the Macintosh, hold down COMMAND, and then select the button or graphic object.
2. From the Tools menu, choose Assign Macro.
3. To assign an existing macro to the button or graphic object, type or select the name of the macro in the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To clear the macro assigned to the button or graphic object, delete the macro name from the Macro Name/Reference box, and then choose the OK button.

   –Or–

   To assign a new macro to the button or graphic object, choose the Record button, and then follow the steps for recording a macro.

The procedure for changing the macro assigned to a custom button on a toolbar is similar, but you must first choose the Toolbars command from the View menu and then choose the Customize button.

Setting and Changing Macro Options

You learned about some of the options you can set when you record a macro in "Recording a Macro" earlier in this chapter. After you have recorded a macro, you can change the following macro options:

- The description of the macro
- The keyboard shortcut that runs the macro
- The name of the macro as it appears on the Tools menu

In addition, you can set options that assign the macro to a topic in a Help file. For more information about using Help files, see Appendix C, "Displaying Custom Help Topics." To learn how to change the name of a macro, see Chapter 2, "Editing Recorded Macros."

▶ **To set or change options for an existing macro**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select the name of the macro whose options you want to change.
3. Choose the Options button.
4. Change any of the options shown.

   For a description of these options, choose the Help button.
5. Choose the OK button.

Controlling the Type of References You Record

One of the options that you can set at the time you record a macro is the type of reference recorded. A *reference* is the address of a cell in Microsoft Excel, such as A1 or B5. A reference is either absolute or relative.

When you set the reference type to *absolute*, Microsoft Excel keeps track of the exact position of each cell you select. When you set the reference type to *relative*, however, Microsoft Excel keeps track of the position of each cell you select relative to the previously selected cell.

For example, consider what happens if you select cell A1 and then record a macro that selects cell C4 and makes it bold. Keep in mind that cell C4 is two cells to the right and three cells down from cell A1, the previous selection.

If you set the reference type to absolute before you select cell C4, the macro always makes cell C4 bold. If you set the reference type to relative before you select cell C4, however, the macro selects the cell that is two cells to the right and three cells down from the active cell and makes it bold. So the macro selects cell C4 if the active cell is A1, D4 if the active cell is B1, E5 if the active cell is C2, and so on.

The advantage of using relative references is that you can run the resulting macro anywhere on your worksheet. With absolute references, on the other hand, you can run your macro on the same cells no matter where you are on the worksheet. For example, the CreateTitle sample macro recorded earlier in this chapter is designed to act on cell C3 every time, so you might choose to record absolute references for this macro instead of relative references.

▶ **To set the reference type**

- From the Tools menu, choose Record Macro, and then choose Use Relative References.

  If the Use Relative References command already has a check mark by it, Microsoft Excel is using relative references. To switch to absolute references, choose the command, which changes the reference type and removes the check mark.

  If the Use Relative References command does not have a check mark by it, Microsoft Excel is using absolute references. To switch to relative references, choose the command, which changes the reference type and adds the check mark.

For more information about absolute and relative references, see Chapter 10, "Creating Formulas and Links," in the *Microsoft Excel User's Guide.*

Making Recorded Macros Available All the Time

Another option you can set when you choose the Options button in the Record New Macro dialog box (which is displayed by choosing Record Macro and then Record New Macro from the Tools menu) specifies where to store the macro you are recording—in the active workbook, in a new workbook, or in the Personal Macro Workbook.

The *Personal Macro Workbook* is a hidden workbook that is always open, unless you specify otherwise. It is a handy depository for recorded macros that you use in a variety of workbooks. Because the Personal Macro Workbook is always open, the macros you store in it are always available. The macro names appear in the Macro dialog box (which is displayed when you choose Macro from the Tools menu), and the macro shortcut keys are available (unless macros in other visible workbooks use the same keys). Microsoft Excel manages the Personal Macro Workbook for you, so you don't have to remember to save it.

Displaying the Personal Macro Workbook

The Personal Macro Workbook works similarly to any other workbook except that it initially contains only one sheet where all the macros you specify are stored. Later, after you have learned more about Visual Basic, you might want to add other items to the Personal Macro Workbook.

To display the Personal Macro Workbook, choose the Unhide command from the Window menu.

Finding the Personal Macro Workbook

The Personal Macro Workbook does not exist until a macro is recorded in it. Then the workbook is stored in the startup directory or folder named PERSONAL.XLS (in Microsoft Excel for Windows) or PERSONAL MACRO WORKBOOK (in Microsoft Excel for the Macintosh). For more information about the startup directory or folder, see Chapter 35, "Controlling What Happens When You Start Microsoft Excel," in the *Microsoft Excel User's Guide.*

Information About Global Macros for Microsoft Excel 4.0 Users

If you are upgrading from Microsoft Excel version 4.0, the global macros you stored in the Global Macro Sheet are still available for Microsoft Excel version 5.0. The file GLOBAL.XLM (in Microsoft Excel for Windows) or GLOBAL MACRO SHEET (in Microsoft Excel for the Macintosh) opens automatically as a hidden document as it did in Microsoft Excel 4.0. However, global macros that you record in Microsoft Excel 5.0 are stored in the Personal Macro Workbook.

Tips for Using the Macro Recorder

Sometimes a recorded macro behaves in ways you don't expect when you try to run it on a new worksheet. The following tips help you record macros to better suit your needs:

- Plan what you want to do before you begin recording.

  Remember that the Macro Recorder records everything you do, including your mistakes.

- Select cells or objects first, and then turn on the recorder.

  This makes your macros portable—that is, applicable to whatever the active selection is when you run the macro. An exception is a macro that you want to run on the same cell every time. (The CreateTitle sample macro you recorded earlier in this chapter is one in which you first turn on the recorder and then select the cell.)

- Switch to the appropriate workbook and select the appropriate sheet before you turn on the recorder.

- View your macros while you are recording them.

  Your Visual Basic macros are stored on special sheets called *Visual Basic modules.* After you have started the recorder, you can choose the Arrange command on the Window menu to view the module while working on your worksheet. With this feature, you can watch the Macro Recorder actually create the Visual Basic code as you work. For more information about Visual Basic modules, see Chapter 4, "Introducing Visual Basic Procedures."

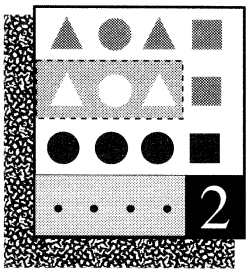
- Use the Macro Recorder as a learning tool.

  When you examine the Visual Basic code that the recorder creates, you'll learn more about how Visual Basic works. (Note, however, that the recorder doesn't always create the most attractive or efficient code, so you'll usually want to edit the code you record; that's why you'll want to read the next chapter.)

This concludes your introduction to macros and the Macro Recorder. Although recording macros is very useful for automating simple tasks, the Macro Recorder does not anticipate exactly what you want it to do in every situation. Most of the time, you'll want to edit and customize recorded macros to suit your needs. The next chapter shows you how to edit the Visual Basic code that the Macro Recorder creates. In later chapters, you'll learn how to write your own Visual Basic code.

CHAPTER 2

Editing Recorded Macros



In Chapter 1, "Automating Repeated Tasks," you learned how to record a macro that automates turning off gridlines and changing border and font formats. This recorded macro does its task well, but it can be enhanced to make it more useful.

This chapter discusses the ways you can view, study, and modify the macros you record, using the CreateTitle macro you recorded in Chapter 1 as an example. You can modify this macro so that it both turns on and turns off gridlines, enters a title each time you run the macro, and sets only the font and border options you select.

Contents

Viewing and Modifying a Recorded Macro

The first step in editing a macro is to display it. Macros are written in the Visual Basic programming language and stored on special sheets called Visual Basic modules. There are three ways to display a macro so you can learn about it or adapt it. First, you can display a macro with the Macro dialog box.

▶ **To display a macro using the Macro dialog box**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select the name of the macro you want to display.
3. Choose the Edit button.

   The macro appears in a window.

You can also display a macro you've assigned to a button or other graphic object.

▶ **To display a macro assigned to a worksheet button or other graphic object**

1. In Microsoft Excel for Windows, click the object with the right mouse button.

   In Microsoft Excel for the Macintosh, hold down COMMAND and click the object.
2. From the shortcut menu, choose Assign Macro.
3. In the Macro Name/Reference box, type or select the name of the macro you want to display.
4. Choose the Edit button.

Finally, if you know the name of the Visual Basic module containing the macro, you can simply switch to that module and display its code.

Example

This example shows you how to display the Visual Basic code that corresponds to the CreateTitle sample macro recorded in Chapter 1.

▶ **To display the CreateTitle macro**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select **CreateTitle**.
3. Choose the Edit button.

When you follow these steps, Microsoft Excel displays a Visual Basic module (Module1, unless you've changed the name of the sheet or inserted a module before recording the macro) that contains the following code. (Your code might look slightly different, depending on the steps you followed in Chapter 1.)

```
' CreateTitle
'
' Sets up a worksheet with a title in cell C3 and applies custom formats
'
Sub CreateTitle()
    ActiveWindow.DisplayGridlines = False
    Range("C3").Select
    ActiveCell.FormulaR1C1 = "West Coast Sales"
    With Selection.Font
        .Name = "Times New Roman"
        .FontStyle = "Bold Italic"
        .Size = 18
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlNone
        .ColorIndex = xlAutomatic
    End With
```

```
    Selection.Borders(xlLeft).LineStyle = xlNone
    Selection.Borders(xlRight).LineStyle = xlNone
    Selection.Borders(xlTop).LineStyle = xlNone
    Selection.Borders(xlBottom).LineStyle = xlNone
    Selection.BorderAround weight:=xlThick, colorIndex:=xlAutomatic
    Selection.EntireColumn.Autofit
End Sub
```

It isn't necessary to completely understand how this macro works now; you can follow the steps in the Example sections in this chapter to get some experience editing macros. However, if you want to know more about a specific term, you can select that term and press the F1 key (in Windows) or COMMAND+SHIFT+QUESTION MARK (on the Macintosh) to display the Help topic for it.

Reading Visual Basic Code

When you record a macro, Microsoft Excel creates a series of statements equivalent to the actions you perform. You can edit these statements to modify the macro, and you can also add comments to clarify what's happening in your code.

Statements

Statements are instructions to Microsoft Excel to perform actions. For example,

```
ActiveCell.FormulaR1C1 = "1994 Sales Projections"
```

is a Visual Basic statement entering the text "1994 Sales Projections" into the active cell. It is equivalent to manually typing the text into the active cell and pressing ENTER, RETURN, or an arrow key to lock in the entry.

Statements can consist of keywords, operators, variables, and procedure calls, as described in Chapter 6, "Working with Visual Basic Code in Procedures."

Keywords are terms that have a special meaning in Visual Basic. For example, the **Sub** and **End Sub** keywords mark the beginning and end of a macro. By default, keywords appear as blue text on color monitors.

Tip To find out more about a keyword used in a macro, simply select the keyword and press F1 (in Windows) or COMMAND+SLASH (on the Macintosh). When you do this, a topic in the online Visual Basic Reference appears and presents more information about the term. If the term is a variable name, comment, or the name of a macro, the Help system presents a window that lists additional topics. You can also search the online Visual Basic Reference for the name of a keyword. The *Microsoft Excel Visual Basic Reference* is available in printed form from Microsoft Press. For ordering information, see this manual's "Introduction."

Comments

Comments describe a macro and how its parts work together, but Microsoft Excel ignores comments when the code runs. Adding comments greatly simplifies the task of adapting macros to new situations, because it's easier to remember what is happening in a given macro if you've described its purpose and operation in your own words. You indicate the beginning of a comment with a single apostrophe, as shown in the first line of the macro displayed in the preceding example. Comments always continue to the end of the line; working code that follows a comment must go on a new line.

Microsoft Excel automatically adds a comment to the beginning of a recorded macro; the comment contains the name of the macro and its description. By default, comments appear as green text on color monitors. (You can change the colors associated with the various parts of a macro, as discussed in "Customizing Visual Basic" later in this chapter.)

Scrolling and Moving the Insertion Point

To scroll manually to a part of a macro that does not appear in the window of the Visual Basic module, you can click the scroll bars at the right and bottom sides of the window. The position of the scroll box indicates the position of the text relative to the rest of the module. When you use the scroll bars, the insertion point does not move from its original location. To edit a macro, you must move the insertion point manually.

To move the insertion point using the mouse, scroll through the window, position the pointer where you want to place the insertion point, and click. In addition to using the mouse, you can use key combinations to move the insertion point; when you do this, the window scrolls automatically.

To move the insertion point	Press this key combination
One word to the left	CTRL+LEFT ARROW
One word to the right	CTRL+RIGHT ARROW
To the beginning of the module	CTRL+HOME
To the end of the module	CTRL+END
To the next sheet	CTRL+PAGE DOWN
To the previous sheet	CTRL+PAGE UP
To the next macro	CTRL+DOWN ARROW
To the previous macro	CTRL+UP ARROW

Splitting the Window into Panes

Sometimes you need to see different parts of a macro at the same time, or two macros in the same Visual Basic module. Rather than repeatedly scrolling up and down in the module, you can split the document into panes. You can scroll independently in each pane, making it easy to compare different parts of a macro or to copy and paste code from one location to another.

▶ **To split the window into two panes with the mouse**

- Double-click the split box on the vertical scroll bar.

  –Or–

  Drag the split box down until the panes appear in the proportions you want.

You can also use the keyboard to split a window into panes.

▶ **To split the window into two panes with the keyboard**

- From the Window menu, choose Split.

Editing Code

You edit code the same way you edit text in most word-processing programs: You select and delete code, type over code, or insert code by moving the insertion point and then typing new text.

To identify the text you want to edit, you select it. For example, to delete a sentence, you select and then delete it. You can select text in a variety of ways with either the mouse or the keyboard.

One technique—dragging over text with the mouse—is a standard way of selecting text. Often, however, the editing changes you make involve whole words or lines of code. You can quickly select the appropriate amount of text with a few keystrokes or clicks of the mouse.

▶ **To select an item or area of text**

- Place the mouse pointer where you want the selection to begin, hold down the mouse button, and drag the pointer to where you want the selection to end.

  You can also place the insertion point where you want the selection to begin and then hold down SHIFT as you click where you want the selection to end. Or you can hold down SHIFT and use the arrow keys to move the insertion point to where you want the selection to end.

If you select too much or too little text, use the following method to adjust the amount of text you selected.

▶ **To adjust a selection**

- Hold down SHIFT and click where you want the selection to end.

  You can also hold down SHIFT and use an arrow key to move to where you want the selection to end.

To cancel a selection, you can click anywhere in the Visual Basic module or press an arrow key.

Adding Comments to Code

Adding comments to code is an excellent way to identify the purpose of the various parts of the macro and to find your way around the macro more easily when you need to adapt the macro to changing circumstances.

Example

You can add some comments to the CreateTitle macro recorded in Chapter 1; for the following actions, refer to the code in the preceding example. Move the insertion point to the end of the `Sub CreateTitle ( )` statement, press ENTER twice, press TAB once, and type the following comment.

```
'Set options from Tools Options View dialog box.
```

Move the insertion point to the end of the statement `Range("C3").Select`, press ENTER, and type this comment.

```
'Enter the title.
```

Move the insertion point to the end of the statement that begins `ActiveCell`, press ENTER twice, and type the following.

```
'Set font formats.
```

Move the insertion point to the end of the next `End With` statement, press ENTER twice, and type the following.

```
'Set border formats.
```

Move the insertion point to the end of the second statement that ends `colorIndex:=xlAutomatic,` press ENTER twice, and type this comment.

```
'Set column width.
```

Deleting Unnecessary Code

The Macro Recorder creates code that corresponds to most actions you take. Often, however, more instructions are recorded in the Visual Basic module than you actually want to appear in the macro, and you need to delete the extra code so that the macro does exactly what you want.

Example

Suppose that you do not want to enter the worksheet title in cell C3 every time you run the CreateTitle macro. To enter and format the title in whichever cell is selected at the time you run the macro, delete the following statement.

```
Range("C3").Select
```

You can also change the title formatting by editing the block that sets text formats. To do this, delete each statement that is not underlined in the following excerpt.

```
'Set font formats.
With Selection.Font
    .Name = "Times New Roman"
    .FontStyle = "Bold Italic"
    .Size = 18
    .Strikethrough = False
    .Superscript = False
    .Subscript = False
    .OutlineFont = False
    .Shadow = False
    .Underline = xlNone
    .ColorIndex = xlAutomatic
End With
```

The CreateTitle macro should now look like the following example.

```
' CreateTitle
'
' Sets up a worksheet with a title in cell C3 and applies custom formats
'
Sub CreateTitle()
    'Set options from Tools Options View dialog box.
    ActiveWindow.DisplayGridlines = False

    'Enter the title.
    ActiveCell.FormulaR1C1 = "West Coast Sales"

    'Set font formats.
    With Selection.Font
        .Name = "Times New Roman"
        .FontStyle = "Bold Italic"
        .Size = 18
    End With

    'Set border formats.
    Selection.Borders(xlLeft).LineStyle = xlNone
    Selection.Borders(xlRight).LineStyle = xlNone
    Selection.Borders(xlTop).LineStyle = xlNone
    Selection.Borders(xlBottom).LineStyle = xlNone
    Selection.BorderAround weight:=xlThick, colorIndex:=xlAutomatic
```

```
    'Set column width.
    Selection.EntireColumn.AutoFit
End Sub
```

You should also edit the comment line that specifically mentions cell C3. The edited version of the macro is considerably shorter and performs only the operations you want. To make sure you haven't deleted any statements inadvertently, run the macro again; it should insert the title in the active cell and format it appropriately.

Finding and Replacing Code

When you need to find and change text in a macro, use the Find and Replace commands on the Edit menu. For example, if you change the name of a worksheet referenced in your macros, you need to change every instance of the name in your code. Use Find to locate occurrences of the text you specify. Use Replace to change instances of the specified text.

▶ **To find text**

1. With a Visual Basic module active, choose Find from the Edit menu.
2. In the Find What box, type the text you want to find.
3. To indicate the direction of your search, select Up or Down in the Search box, or select All to search the entire selection.
4. To indicate the scope of your search, select Procedure, Module, All Modules, or Selected Text in the Look In box.
5. To indicate how you want your search to proceed, select any of the other options, as shown in the following table.
6. Choose the Find Next button.

To specify text you want to find, use the following options in the Find dialog box.

To	Select
Find whole words, not characters inside words — for example, "act" but not "acting"	Find Whole Words Only
Find words with a certain pattern of uppercase and lowercase letters — for example, "Act" but not "act"	Match Case
Find words with a certain pattern of letters — for example, to use **Act*** to search for "Actor," "Acting," and so on	Use Pattern Matching

When you choose the Find Next button, Microsoft Excel selects the next occurrence of the text or displays a message saying that the text hasn't been found. To find the next occurrence of the text, choose the Find Next button again.

To edit the selected text, choose the Cancel button. To continue the search after editing the found text, choose Find from the Edit menu, and repeat the steps for finding text.

If you begin the search from the middle of the search area you specified in step 4, Microsoft Excel searches from the position of the insertion point to either the end or beginning of the macro and then displays a message asking if you want to continue the search. Choose the OK button to search through the remainder of the search area, or choose the Cancel button to stop searching.

To replace text

1. From the Edit menu, choose Replace.
2. In the Find What box, type the text you want to find.
3. In the Replace With box, type the replacement text.
4. To indicate the direction of your search, select Up or Down in the Search box, or select All to search the entire selection.
5. To indicate the scope of your search, select Procedure, Module, All Modules, or Selected Text in the Look In box.
6. To indicate how you want your search to proceed, select any of the other options, as shown in the preceding table.
7. To confirm each change, choose the Replace button.

   –Or–

   To change all occurrences of the text without confirmation, choose the Replace All button.

You can also cancel or undo the effect of the last replacement.

To cancel or undo replacing

- From the Edit menu, choose Undo.

  Only the last change is reversed if you confirmed each change separately (that is, if you chose the Replace button). All changes are reversed if you made the changes in one step (that is, if you chose the Replace All button).

Moving and Copying Code

The following steps show how to move and copy code using the Clipboard. Use the Clipboard if you want to temporarily store the code you move or copy.

▶ **To move or copy code using the Clipboard**

1. Select the code you want to move.
2. To move the code, choose Cut from the Edit menu, or press CTRL+X.

   –Or–

   To copy the code, choose Copy from the Edit menu, or press CTRL+C.
3. Position the insertion point where you want to insert the text.

   If you want to copy the code to another Visual Basic module, you can also switch to the module and then position the insertion point.
4. From the Edit menu, choose Paste, or press CTRL+V.

Example

The CreateTitle macro actually performs two distinct tasks: It sets display options for the active worksheet, and it enters text into and formats a cell on that worksheet. By moving and copying code from one location to another, you can divide these tasks so that you can use them independently of each other. Dividing a large task into smaller tasks also makes it easier to understand and maintain your macros.

To divide the CreateTitle macro into two more manageable macros, select the following statements, and then choose Cut from the Edit menu.

```
    'Set options from Tools Options View dialog box.
    ActiveWindow.DisplayGridlines = False
```

Place the insertion point after the `End Sub` statement, press ENTER twice, and type the following.

```
Sub SetGridlines()
```

Make sure you press ENTER at the end of the line. Next, to insert the code in its new location, choose Paste from the Edit menu, press ENTER, and then type the following on the next line.

```
End Sub
```

At this point, there are two macros: CreateTitle and SetGridlines. The corresponding code should look similar to the following example.

```
Sub CreateTitle()
    'Enter the title.
    ActiveCell.FormulaR1C1 = "West Coast Sales"

    'Set font formats.
    With Selection.Font
        .Name = "Times New Roman"
        .FontStyle = "Bold Italic"
        .Size = 18
    End With
```

```
    'Set border formats.
    Selection.Borders(xlLeft).LineStyle = xlNone
    Selection.Borders(xlRight).LineStyle = xlNone
    Selection.Borders(xlTop).LineStyle = xlNone
    Selection.Borders(xlBottom).LineStyle = xlNone
    Selection.BorderAround weight:=xlThick, colorIndex:=xlAutomatic

    'Set column width.
    Selection.EntireColumn.AutoFit
End Sub

Sub SetGridlines()
    'Set options from Tools Options View dialog box.
    ActiveWindow.DisplayGridlines = False
End Sub
```

The only step remaining is to assign the new SetGridlines macro to a key combination, button, or graphic object.

▶ **To assign the SetGridlines macro to a key combination**

1. From the Tools menu, choose Macro.
2. In the Macro Name/Reference box, type or select **SetGridlines**.
3. Choose the Options button.
4. Under Assign To, select the Shortcut Key check box, and type **G** in the box.
5. Choose the OK button and then the Close button.

Recording Code into an Existing Macro

Recording code into an existing macro is useful when you've recorded and edited a lengthy sequence of steps and then decide that one of the tasks needs to be changed or a new task needs to be included.

To add code to an existing macro, you can manually type the code in the Visual Basic module containing the macro. However, a quick way to add new tasks to an existing macro is to record new code that Microsoft Excel inserts at the location you specify.

▶ **To record and insert new code in an existing macro**

1. Place the insertion point in the macro where you want to add code.
2. From the Tools menu, choose Record Macro, and then choose Mark Position For Recording.
3. Switch to the sheet from which you want to record the new tasks.
4. From the Tools menu, choose Record Macro, and then choose Record At Mark.

5. Perform the tasks you want to record.
6. Click the Stop Macro button.

   You can also choose Record Macro from the Tools menu and then choose Stop Recording.

Inserting Text from a File

It's often helpful to store code in a file other than your workbook, so that you can edit it in a word processor such as Microsoft Word or keep a backup copy of your code stored as text. Alternatively, you might have been given a text file downloaded from an information service and might now want to transfer the code to a Visual Basic module.

To insert text from a file

1. With a Visual Basic module active, place the insertion point where you want to insert text from a file.
2. From the Insert menu, choose File.
3. In the File Name box, type or select the name of the file containing the text you want to insert.
4. Choose the OK button.

Enhancing a Macro with Visual Basic Features

Visual Basic offers much more power than is available to you through recording macros alone. Using Visual Basic, you can test your macros for special conditions and perform an action only when a certain condition has been met. Your macros can request information from the user or notify the user of the progress of the macro.

The following sections show you how to add some of the special features of Visual Basic to your macros. It isn't as important at this point to understand every element of the examples presented here as it is to see that you can make recorded macros more flexible by using features of Visual Basic that cannot be recorded in a macro.

Changing the State of an Option

Example

As written, the SetGridlines macro described earlier in this chapter only turns off gridlines. You can write another macro to turn the display of gridlines on, but you can also modify SetGridlines so that it *toggles* the option: Pressing the key combination you assigned the macro turns off gridlines, and pressing it again turns gridlines on.

You can find the current state of the Gridlines setting in the Options dialog box (available when you choose the Options command on the Tools menu) and use the **Not** keyword to toggle the state of the option. Edit the SetGridlines macro by changing this line in the preceding code example

```
ActiveWindow.DisplayGridlines = False
```

to match the underlined statement in the following.

```
Sub SetGridlines( )
    'Set options from Tools Options View dialog box.
    ActiveWindow.DisplayGridlines = Not ActiveWindow.DisplayGridlines
End Sub
```

Making a Macro Interactive

Example

As written, the CreateTitle macro always enters the text "West Coast Sales" into the active cell. Another way to make the CreateTitle macro more useful is to let the user enter the title at the time when the macro runs and store this information in a *variable,* which serves as a temporary holding place for the information. (For more information about using variables, see Chapter 4, "Introducing Visual Basic Procedures.")

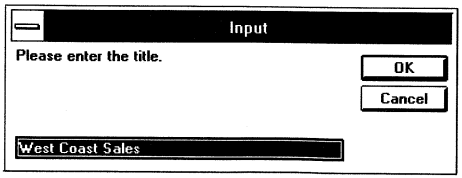
To create this input box, change the lines

```
'Enter the title.
ActiveCell.FormulaR1C1 = "West Coast Sales"
```

to read

```
'Enter the title.
TitleText = InputBox( _
    prompt := "Please enter the title.", _
    default := "West Coast Sales")
ActiveCell.FormulaR1C1 = TitleText
```

When the CreateTitle macro runs, Microsoft Excel displays the following dialog box.



Adding Control Structures

You can make a macro more powerful by adding control structures. With control structures, you can instruct Microsoft Excel to repeat statements and to make decisions about which statements to run.

For example, assume you want to make bold every cell in the range A1:A10 that contains a number greater than 500. You cannot accomplish this task by simply recording a macro because there is no menu command in Microsoft Excel that you can use to compare the value of a cell with 500 or to repeat an action 10 times.

However, you can use two control structures — an **If...Then** statement and a **For...Next** loop — to accomplish the task, as shown in the following code.

```
'Makes bold cells that contain a number greater than 500.
Sub BoldCells ()
    For I = 1 To 10                        'For each cell in range,
        If Cells(I, 1).Value > 500 Then  'If greater than 500,
            Cells(I, 1).Font.Bold = True 'make its text bold.
        End If
    Next I
End Sub
```

For more information about using control structures in Visual Basic code, see Chapter 7, "Controlling How Your Code Runs."

Copying a Code Example from Help

Many of the topics in the online Visual Basic Reference contain code examples that you can copy and paste into a Visual Basic module. Some of the examples are complete macros; others consist of just one or two lines of code.

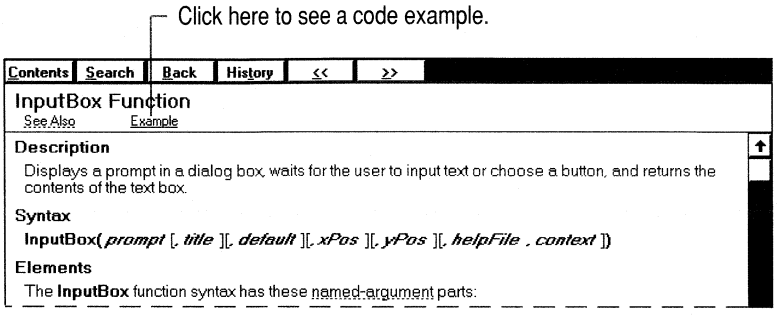
▶ **To copy a code example from Help**

1. With a Visual Basic module active, place the cursor in the Visual Basic keyword for which you want an example.
2. In Microsoft Excel for Windows, press F1. In Microsoft Excel for the Macintosh, press COMMAND+/.

   The Help topic for the keyword appears.
3. At the top of the Help window, click the word "Example," as shown in the following illustration.
4. In the Example window, choose the Copy button.

   Help displays a Copy dialog box.
5. Select the portion of the example you want to copy.
6. Choose the Copy button.

7. Return to your Visual Basic module.
8. Position the insertion point where you want to paste the code.
9. From the Edit menu, choose Paste.



Printing a Visual Basic Module

You can print every macro in an active Visual Basic module; this is useful for reviewing, rewriting, and debugging a system of macros while you're away from the computer.

▶ **To print code**

1. With a Visual Basic module active, choose Print from the File menu.
2. Set the printing options you want.
3. Choose the OK button.

Customizing Visual Basic

You can customize three categories of options in Visual Basic — the general environment, the code element colors, and the international settings.

▶ **To customize Visual Basic options**

1. From the Tools menu, choose Options.
2. Select the Module Format or Module General tab.
3. Change any of the options shown.
4. Choose the OK button.

The following table describes the options you can set on the Module Format tab.

Option	Description
Font	The font of the text displayed in the module and Debug window
Size	The point size of the text displayed in the module and Debug window
Code Colors	The background and foreground colors for various code elements

The following table describes the options you can set on the Module General tab.

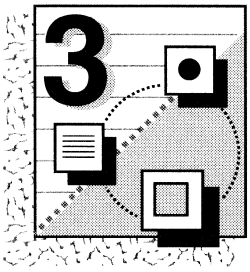
Option	Description
Auto Indent	Whether code is automatically indented by the same amount as the previous line.
Tab Width	The width of tab stops in spaces (from 1 to 40).
Display Syntax Errors	Whether each line of code you type is immediately checked for correct syntax.
Break On All Errors	Whether break mode is entered when Microsoft Excel encounters an error in the code. When this option is set, Microsoft Excel enters break mode in this situation. When it is not set, Microsoft Excel calls a user-defined error handler if one exists.
Require Variable Declaration	Whether each variable must be declared explicitly.

Finally, you can set certain international options in the Options dialog box. For more information about international options, see Appendix A, "Writing Code for International Use."

For more information about variables and declarations, see Chapter 6, "Working with Visual Basic Code in Procedures." For information about break mode and user-defined error handlers, see Chapter 8, "Testing and Debugging Your Code."

CHAPTER 3

Creating User-Defined Functions



This chapter describes how to design and create user-defined functions and enter them in a worksheet. A user-defined function is like any built-in Microsoft Excel worksheet function, such as SUM or AVERAGE. Because you create the user-defined function, however, you decide exactly what it does.

A single user-defined function can often replace a long or nested worksheet formula or even a set of formulas. Replacing several worksheet formulas with one is part of what makes user-defined functions easier to remember and more efficient to use.

Contents

All user-defined functions use some Visual Basic code, but you don't have to know much about programming to put user-defined functions to work. This chapter is just a starting point. It introduces you to user-defined functions but avoids many terms, topics, and pieces of information that are important in Visual Basic.

Still, the more you learn about Visual Basic, the better able you'll be to write functions that suit your needs. For more detailed information about user-defined functions, see Chapter 6, "Working with Visual Basic Code in Procedures," and Chapter 7, "Controlling How Your Code Runs."

What a User-Defined Function Does

You create a user-defined function in a Visual Basic module by combining mathematical expressions, built-in Microsoft Excel functions, and Visual Basic code. You supply your user-defined function with a set of values; it performs calculations on those values and returns a new value.

Note A user-defined function can also work with text, dates, and values, not just numbers and mathematical expressions. You will learn more about these types of user-defined functions in Chapter 6.

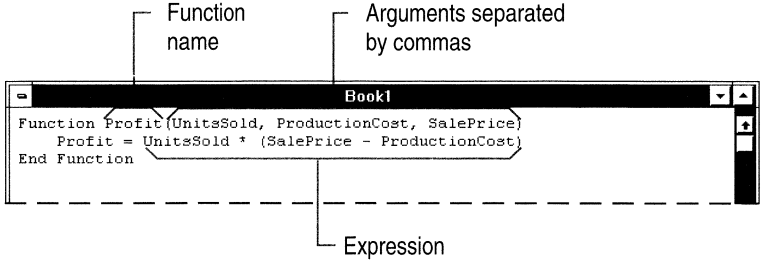
A user-defined function is similar to a macro. But there are differences between a user-defined function and the type of macro you've learned to record and edit so far. Some of these differences are listed in the following table.

Recorded macros	User-defined functions
Perform an action, such as creating a chart or moving cells	Return a value; cannot perform actions
Can be recorded	Must be created in a Visual Basic module
Are enclosed by the keywords **Sub** and **End Sub**	Are enclosed by the keywords **Function** and **End Function**

The most important difference to remember is that recorded macros perform actions that change a sheet in some way. User-defined functions return values.

The Parts of a User-Defined Function

The following illustration shows the parts of a very simple user-defined function. The Profit function calculates gross profit based on the number of units sold of a particular product, the total cost to manufacture each product, and the price obtained for each unit. The function has three arguments (UnitsSold, ProductionCost, and SalePrice) and one mathematical expression.



Since a user-defined function accepts values, performs calculations, and returns a value, you'll need to design the following parts to handle these operations in your user-defined functions:

- The **Function** and **End Function** statements. These Visual Basic keywords mark the beginning and the end of the function.
- A name. This is the unique identifier of the function.

- Arguments. These are the values you supply—the starting point from which your function makes its calculations. To define arguments to a user-defined function, type the argument names in parentheses following the function name, separating each argument from the others with a list separator.

  The list separator character varies depending on your country settings. In English, the separator is a comma. For more information about writing code for international use, see Appendix A, "Writing Code for International Use."
- Visual Basic code and expressions. These are the instructions that tell the user-defined function which calculations to perform. An *expression* is combination of numbers, variables, and mathematical operators that yields a value.
- The return value. This is the value that the user-defined function returns after performing its calculations. The way you specify a return value is to follow the name of the user-defined function with an expression.

  NameOfFunction = expression

Understanding the Parts of a User-Defined Function

Some parts of a user-defined function are similar to items you've already worked with on worksheets, while other parts might be new to you. This section builds on your existing knowledge about worksheets and worksheet formulas and explains each of the parts. For background information on functions, see Chapter 10, "Creating Formulas and Links," in the *Microsoft Excel User's Guide.*

Arguments

Arguments in user-defined functions are similar to names that you define for cells on worksheets. An argument is a type of variable. A *variable* is a name that represents a value; it can also be a reference to an object such as a cell or worksheet. Just as names make your worksheet formulas easier to read and understand, arguments and variables are the basis of comprehensible expressions in your user-defined functions.

Example

The following function calculates after-tax profit (net profit). Its arguments, shown here, are UnitsSold, ProductionCost, SalePrice, and IncomeTaxRate. The other variables are `GrossProfit` and `NetProfit`. `NetProfit` is a special variable because it contains the return value.

```
Function NetProfit(UnitsSold, ProductionCost, SalePrice, IncomeTaxRate)
    GrossProfit = UnitsSold * (SalePrice - ProductionCost)
    NetProfit = GrossProfit * (1 - IncomeTaxRate)
End Function
```

This example shows you how arguments and other variables work together in expressions. The expression that calculates gross profit uses three of the function's arguments; the result of the expression is stored in the variable `GrossProfit`. The expression that calculates net profit uses one argument and the variable `GrossProfit`; the result is stored in `NetProfit`.

You can create a user-defined function that does not require arguments. The function could, for example, use the value in a cell or the time of day as a starting point from which to make calculations, instead of using values you supply in arguments. RAND and NOW are built-in worksheet functions that do not require arguments.

Visual Basic Expressions

Expressions are similar to the formulas you enter on worksheets. Most of the functions and mathematical operators that you use in worksheet formulas are available to user-defined functions. But there are several differences between worksheet formulas and Visual Basic expressions:

- While worksheet formulas are entered in cells, you enter Visual Basic expressions in a Visual Basic module.
- Worksheet formulas start with an equal sign, and the result is put in the cell containing the formula. Because Visual Basic needs to know where to put the result of a user-defined function, however, Visual Basic expressions are preceded by a variable and an equal sign. The variable on the left side of the equation is the destination or storage site of the value that is calculated on the right. In the following line, `GrossProfit` is a variable containing the result of the mathematical expression to its right.

  ```
  GrossProfit = UnitsSold * (SalePrice - ProductionCost)
  ```

  The complete line of code—a variable followed by an equal sign followed by an expression—is called an *assignment statement* because it assigns a value to a variable.
- You can include Visual Basic statements such as **If**, **For**, and **Do** in user-defined functions.

Example

The NetProfit function in the earlier example illustrates some of the differences between worksheet formulas and Visual Basic expressions. The following user-defined function also shows these differences and includes an **If...Then...Else** statement. The function calculates the commission that a brokerage firm receives from the sale of stock. The commission rate varies depending on the total sale price.

The normal commission rate is $25 plus three cents per share. For transactions greater than $15,000, however, a 10 percent volume discount is provided, so only 90 percent of the shares sold are included in the commission calculation. The function takes into account these variations with the **If...Then...Else** statement.

```
Function Commission(SharesSold, PricePerShare)
    TotalSalePrice = SharesSold * PricePerShare
    If TotalSalePrice <= 15000 Then
        Commission = 25 + 0.03 * SharesSold
```

```
    Else
        Commission = 25 + 0.03 * (0.9 * SharesSold)
    End If
End Function
```

The **If...Then...Else** statement is described in Chapter 7, but you can see how it works in the Commission function. This example is a little more complex than the previous examples in this chapter, but it demonstrates how adding Visual Basic code to your user-defined functions can make them more flexible.

Return Values

The way you specify a return value is to follow the name of the user-defined function with a value or mathematical expression. One or more lines in the function must be in the form *NameOfFunction = expression*. Within a user-defined function, the name of the function acts like a variable. The return value of the function is the value that is stored in *NameOfFunction* when Microsoft Excel finishes calculating the function.

For example, the Commission function includes two different assignment statements that calculate a return value. `Commission = 25 + 0.03 * SharesSold` is used if the total sale price is less than or equal to $15,000; otherwise, `Commission = 25 + 0.03 * (0.9 * SharesSold)` is used. The code is written so that only one of these conditions can be true at a time. After the function performs the calculations, it returns the value stored in the `Commission` variable.

In general, you should design your functions to return one value for a given set of arguments (an exception is a user-defined function that returns an array, which you will learn about in Chapter 6).

Creating a User-Defined Function

In Chapter 2, "Editing Recorded Macros," you learned how to edit previously recorded macros in a Visual Basic module. Now you will write a new user-defined function in a Visual Basic module.

▶ **To create a user-defined function**

1. To switch to a Visual Basic module in the active workbook, select the tab of a Visual Basic module.

   –Or–

   To create a new Visual Basic module, choose Macro from the Insert menu, and then choose Module.

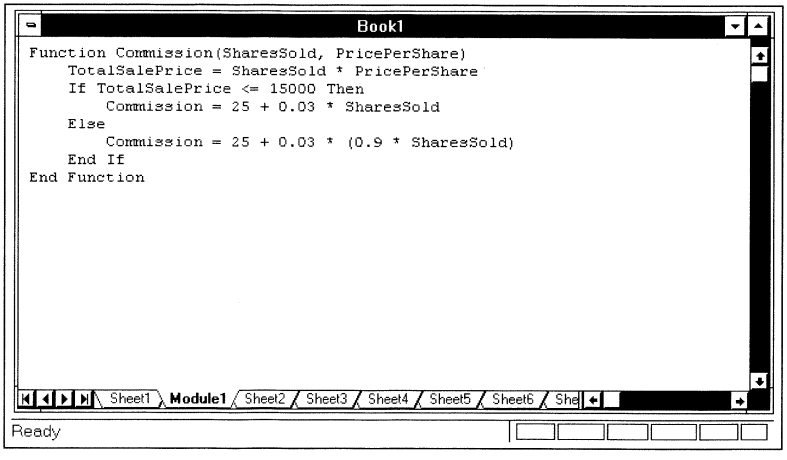
2. Type **Function** followed by the name of your user-defined function.

3. Type the argument list, enclosed in parentheses, separating each argument from the others with a comma or the appropriate list separator.
4. Press ENTER to move to a new line.

   Microsoft Excel checks the syntax of the line you just typed. The color of Visual Basic keywords changes to blue or a previously specified color.
5. Press TAB, type your first line of code, and then press ENTER.

   Pressing TAB indents your code so that it is easier to read.
6. Type the remaining lines of code.
7. Type **End Function** and then press ENTER.

The following illustration shows the Commission function from the previous section in a Visual Basic module.



Putting Your User-Defined Function to Work

Once you have created a user-defined function, you can enter it in a worksheet. A user-defined function is entered and used in exactly the same way as a built-in Microsoft Excel function. In a worksheet cell, for example, you enter the name that you gave your user-defined function just as you would enter the name of a built-in function such as SUM or AVERAGE.

▶ **To enter a user-defined function in a worksheet**

1. In the cell in which you want to enter the function, type an equal sign (=) followed by the name of the function and an opening parenthesis.
2. Type the values, separating each with a comma or the appropriate list separator, followed by a closing parenthesis.
3. Press ENTER.

Note You can enter your function in a selected cell by choosing the Function command from the Insert menu, and then following the instructions in the Function Wizard. For more information about the Function Wizard, see "Using the Function Wizard " in Chapter 10 in the *Microsoft Excel User's Guide.*

The following formula uses the user-defined Commission function shown in the previous illustration. The values for `SharesSold` and `PricePerShare` are 100 and $50, respectively. You would enter this formula in a cell on your worksheet:

```
=Commission(100,50)
```

Making Your User-Defined Functions Easier to Find

In a worksheet, when you choose the Function command from the Insert menu, the Function Wizard dialog box shows you a list of the available functions and their categories. All the math functions, date and time functions, database functions, and so on, are grouped together so that you can find them more easily. After you create a user-defined function, you might find it helpful to assign it to a category. You can use one of the built-in categories or the User-Defined category.

You assign a function to a category with the Object Browser command on the View menu. Some of the terms in the Object Browser dialog box might be unfamiliar to you. You'll learn more about the Object Browser in Chapter 4, "Introducing Visual Basic Procedures."

▶ **To assign a user-defined function to a category**

1. Switch to the Visual Basic module in your workbook that contains the user-defined function.
2. From the View menu, choose Object Browser.
3. In the Libraries/Workbooks box, type or select the name of the workbook containing your module.
4. In the Objects/Modules box, select the module containing your function.
5. In the Methods/Properties box, select the name of your function.
6. Choose the Options button.

7. In the Function Category box, select the name of a category.
8. Choose the OK button.
9. Choose the Show button.

Learning More About User-Defined Functions

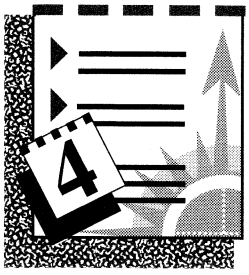
This completes your introduction to user-defined functions. You've seen how Visual Basic statements can enhance user-defined functions—compare the simple Profit function earlier in this chapter to the elaborate decisions made in the Commission function later in this chapter—but you've just begun to learn about Visual Basic itself.

If you will be taking advantage of user-defined functions regularly, you are encouraged to read Chapters 6 and 7. These chapters provide detailed information about passing information to and receiving information from Visual Basic procedures (a user-defined function is a type of procedure) and about the Visual Basic language. You will also learn how to carefully control types of arguments, variables, and return values. This is useful for text functions, date and time functions, and so on.

In the next chapter, you will learn how several macros and user-defined functions can work together to automate complex tasks.

CHAPTER 4

Introducing Visual Basic Procedures

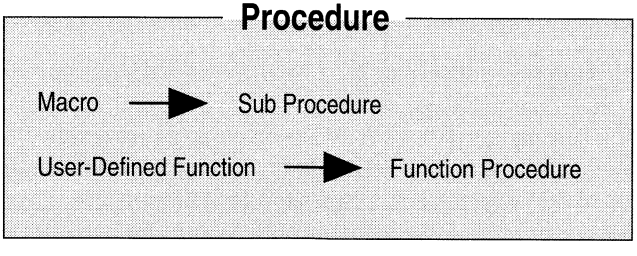


This chapter introduces Visual Basic procedures—what they are, what you do with them, how they work together, and what's inside them. You've actually seen procedures in previous chapters, but they weren't called procedures there. The macros you learned to record in Chapter 1, "Automating Repeated Tasks," are **Sub** procedures, and the user-defined functions you learned to create in Chapter 3, "Creating User-Defined Functions," are a type of **Function** procedure. This chapter explains these types of procedures. In later chapters, you'll learn more about the Visual Basic code that makes procedures work.

Contents

What Is a Visual Basic Procedure?

Now that you're comfortable with macros and user-defined functions, you're ready to use the standard Visual Basic term for these words—procedure.



A *procedure* is a block of Visual Basic code that is entered in a Visual Basic module and is executed as a unit. It is a sequence of Visual Basic statements that does one task well. Visual Basic has two main types of procedures—**Sub** procedures and **Function** procedures.

A ***Sub** procedure* performs an action but does not return a value. It is enclosed by the **Sub** and **End Sub** statements.

```
'Sounds a tone a number of times specified by NumBeeps
Sub MultiBeep(NumBeeps)
    For Counter = 1 To NumBeeps
        Beep                            'Sounds a tone
    Next Counter
End Sub
```

If a **Sub** procedure is run from a Microsoft Excel worksheet or chart sheet (for example, by choosing the Macro command from the Tools menu or by clicking a macro button), it can also be called a macro, but it is still just a **Sub** procedure. If a **Sub** procedure is run only by other procedures in a Visual Basic module, the term macro isn't used. You will learn how one procedure can run another procedure later in this chapter.

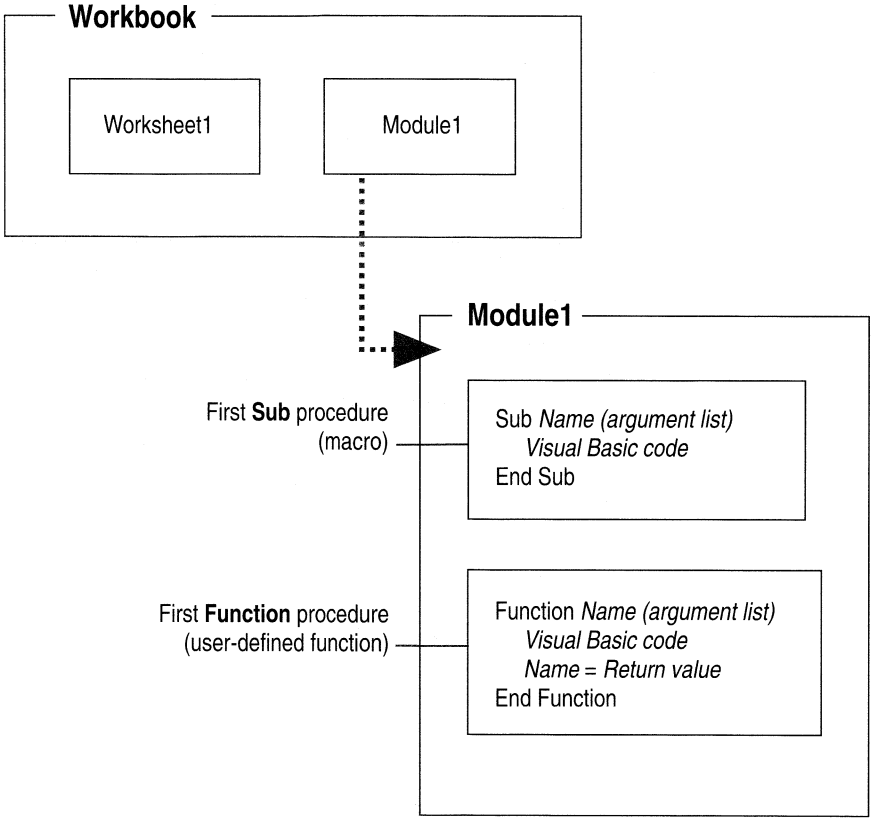
A ***Function** procedure* is similar to a **Sub** procedure, but it returns a value. It is enclosed by the **Function** and **End Function** statements.

```
'Returns the square root of a number, even if the number is negative
Function SafeSqr(Num)
    TempVal = Abs(Num)              'Makes Num positive
    SafeSqr = Sqr(TempVal)          'Returns the square root
End Function
```

If you write your **Function** procedure so that it avoids performing actions that alter the Microsoft Excel environment, you can enter the **Function** procedure name into a worksheet cell and return a value to that cell. In this situation, the **Function** procedure is also called a user-defined function, as you learned in Chapter 3, "Creating User-Defined Functions," but it is still just a **Function** procedure with certain restrictions. If a **Function** procedure is run only by other procedures in a Visual Basic module, the term user-defined function isn't used.

If a **Function** procedure is to be used as a user-defined function, it cannot perform actions that change the Microsoft Excel environment. Examples of these actions include inserting, deleting, or formatting cells or changing cell values; moving, renaming, deleting, or adding sheets or workbooks; and changing the calculation mode or screen views. Such a **Function** procedure is also not allowed to set properties or execute most methods, although it can retrieve properties. In general, you will avoid most problems if your user-defined functions just make calculations based on information that you provide or that can be retrieved from Microsoft Excel.

The following illustration shows how **Sub** procedures (macros) and **Function** procedures (some of which are user-defined functions) are contained within modules, and how modules are contained within workbooks.



There is a third, less common procedure called a *property procedure,* which can include the **Property Get**, **Property Let**, or **Property Set** statement. You will learn about property procedures in "Learning More About Visual Basic Programming" in Chapter 6.

Parts of a Procedure

The parts of a procedure are the same as the parts of a macro described in Chapter 2, "Editing Recorded Macros," and of a user-defined function described in Chapter 3, "Creating User-Defined Functions." The parts of a procedure are:

- The **Function** and **End Function** or **Sub** and **End Sub** statements. These Visual Basic keywords mark the beginning and the end of the procedure.
- A name. This is the unique identifier of the procedure.

- Arguments. These are the values you supply to the procedure—the starting point from which a **Function** procedure makes its calculations, or the information a **Sub** procedure uses to determine what to do and how to do it. You define arguments for a **Sub** or **Function** procedure the same way you define them for a user-defined function—by typing the argument names in parentheses and separating each one from the next with the appropriate list separator.

  ```
  Sub SwapCells (CellRef1, CellRef2)
      Temp = CellRef1.Value
      CellRef1.Value = CellRef2.Value
      CellRef2.Value = Temp
  End Sub
  ```

  The list separator character varies depending on your country settings. In English, the separator is a comma. For more information about the list separator, see Appendix A, "Writing Code for International Use."
- Visual Basic code. This code is the instructions that tell the procedure what to do—the steps or actions to be taken.
- The return value. A **Function** procedure has a return value, but a **Sub** procedure does not.

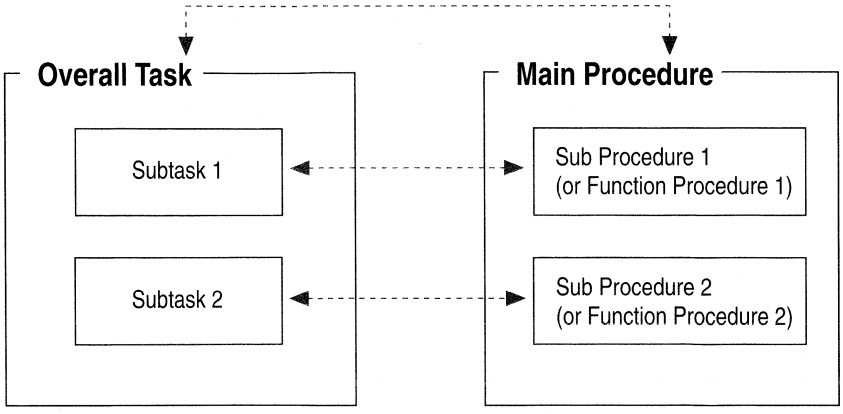
You will learn more about Visual Basic code in procedures in later chapters.

Visual Basic code consists of keywords that you organize in a meaningful way. For information about a particular keyword, search the online Visual Basic Reference for the keyword name, or select the keyword in a Visual Basic module and press F1 (in Windows) or COMMAND+SLASH (on the Macintosh). The *Microsoft Excel Visual Basic Reference* is also available in printed form from Microsoft Press. For ordering information, see this manual's "Introduction."

How Procedures Work Together to Complete a Complex Task

In Chapter 1, "Automating Repeated Tasks," you used the Macro Recorder to create a single **Sub** procedure to automate a simple task. As you learn more about Visual Basic procedures, the tasks you'll want them to complete will become more complex than a single procedure can efficiently complete.

For example, suppose you want to create a procedure to calculate profits after stock is sold. This might seem to be a single task that can be handled by a single procedure, but you might find that it consists of several subtasks—perhaps setting up a new worksheet, getting information from the user about costs and number of shares, and inserting calculated values into your worksheet. Through examination of a task, you can often break it into subtasks, each of which can be handled by a separate procedure.



The problem

The Visual Basic solution

Ideally, a procedure should do one thing well. Your goal as a programmer is to create short, reusable, easy-to-maintain procedures that work together. Short procedures are easier to design, implement, debug, and update because they consist of so much less code than one big procedure does. With multiple procedures, you can also avoid writing the same code several times—just put the common code in one procedure and have other procedures call it as necessary.

A procedure can call, or be called by, another procedure. To *call* a procedure means to run it. The ability of one procedure to call another, or of several procedures to call the same procedure, is what makes them able to work together.

For example, the following procedure, named StockSale, calls three other procedures. The first called procedure, SetUpSheet, sets up a worksheet with heading and formatting information. The second procedure, GetStockData, displays dialog boxes asking the user for information about stock transactions. The third procedure, InsertValues, calculates commissions, taxes, and profits based on the user-provided information and inserts these values into a worksheet.

```
Sub StockSale()
    SetUpSheet
    GetStockData
    InsertValues
End Sub
```

You can continue breaking subtasks into smaller, more manageable sub-subtasks as necessary. Or you might find that your original overall task is a subtask of an even greater task. For example, the StockSale **Sub** procedure calls three other **Sub** procedures, and those three procedures can call, or be called by, other procedures, and so on. Even StockSale itself can be called by another, still larger procedure.

Following is a listing of all the **Sub** procedures called by the StockSale procedure. You might not be able to understand all the code yet, but you can see how each procedure contributes a discrete amount of work to complete its overall task. The procedures are organized in a module; you will learn more about organizing procedures in a module later in this chapter.

```
'Top of module

'Declare module-level variables
Dim OriginalPrice
Dim NumberOfShares
Dim SalePrice

'Declare constants
Const GAINTAXRATE = 0.28
Const HIPROFITS = 1000000
Const SURTAXRATE = 0.1

'Begin procedures
Sub StockSale()
    SetUpSheet      'Calls the SetUpSheet procedure
    GetStockData    'Calls the GetStockData procedure
    InsertValues    'Calls the InsertValues procedure
End Sub

'Prepares a new worksheet by entering formulas, adjusting column widths,
'and turning off gridlines.
Sub SetUpSheet()
    Range("A2").Formula = "Original Price"
    Range("A3").Formula = "Number of Shares"
    Range("A4").Formula = "Sale Price"
    Range("A5").Formula = "Commission"
    Range("A6").Formula = "Taxes"
    Range("A7").Formula = "Net Gain"
    Range("A7").EntireColumn.AutoFit
    Range("B2:B7").NumberFormat = "$#,##0.00_);($#,##0.00)"
    Range("B3").NumberFormat = "0"
    Range("A7:B7").Borders(xlTop).LineStyle = xlSolid
    ActiveWindow.DisplayGridlines = False
End Sub

'Prompt user for information about the transaction. Store responses in
'module-level variables.
Sub GetStockData()
    OriginalPrice = InputBox("What was the purchase price?")
    NumberOfShares = InputBox("How many shares did you buy?")
    SalePrice = InputBox("What was the sale price?")
End Sub
```

```
'Insert user-provided information into cells on the worksheet
Sub InsertValues()
    Range("B2").Value = OriginalPrice
    Range("B3").Value = NumberOfShares
    Range("B4").Value = SalePrice
    MyCommission = Commission(NumberOfShares, SalePrice)
    Range("B5").Value = MyCommission
    'Procedure to look up tax in a tax table.
    MyTax = Tax(NumberOfShares, OriginalPrice, SalePrice)
    Range("B6").Value = MyTax
    GrossProfit = NumberOfShares * (SalePrice - OriginalPrice)
    NetProfit = GrossProfit - MyCommission - MyTax
    Range("B7").Value = NetProfit
End Sub

'Calculate brokerage commission for the transaction
Function Commission(SharesSold, PricePerShare)
    TotalSalePrice = SharesSold * PricePerShare
    If TotalSalePrice <= 15000 Then
        Commission = 25 + 0.03 * SharesSold
    Else
        Commission = 25 + 0.03 * (0.9 * SharesSold)
    End If
End Function

'Compute the capital gains tax
Function Tax(Shares, BuyPrice, SellPrice)
    Profit = Shares * (SellPrice - BuyPrice)
    If Profit <= 0 Then
        Tax = 0
    ElseIf Profit > HIPROFITS Then
        Tax = (GAINTAXRATE * Profit) + (Profit - HIPROFITS) * SURTAXRATE
    Else
        Tax = Profit * GAINTAXRATE
    End If
End Function

'End of module
```

The previous example shows how several **Sub** procedures can work together by calling each other. You can also have **Function** procedures that call each other, **Sub** procedures that call **Function** procedures, and **Function** procedures that call **Sub** procedures.

For example, consider a variation of the StockSale **Sub** procedure that is written as a **Function** procedure. This StockSale procedure calls two other **Function** procedures to calculate a final result.

```
Function StockSale(OriginalPrice, NumberOfShares, SalePrice)
    'GrossProfit is a variable containing the gross profit
    GrossProfit = NumberOfShares * (SalePrice - OriginalPrice)
    'Calls the Commission Function procedure to calculate the commission
    MyCommission = Commission(NumberOfShares, SalePrice)
    'Calls the Tax Function procedure to calculate the taxes
    MyTax = Tax(NumberOfShares, OriginalPrice, SalePrice)
    'Returns the profit after subtracting expenses
    StockSale = GrossProfit - MyCommission - MyTax
End Function
.
.
.
```

The code for the Commission and Tax **Function** procedures would follow, as shown in the previous example.

Note Because none of these **Function** procedures performs actions that improperly alter a worksheet, you can also enter any of their names into a worksheet cell as a user-defined function.

How You Call Procedures

The preceding sections contain examples of procedures calling other procedures. Calling a procedure is similar to running a macro (which you learned about in Chapter 1, "Automating Repeated Tasks"), but the call takes place within Visual Basic code, instead of you running the procedure yourself by clicking a button or choosing a command from a menu.

Each time a procedure is called, Visual Basic executes the statements between that procedure's **Sub** and **End Sub** or **Function** and **End Function** statements. When you call a procedure that has arguments, Visual Basic substitutes the values that you supply in the argument list for the corresponding argument names in the called procedure. For example, suppose you define the MultiBeep procedure as:

```
Sub MultiBeep (NumBeeps)
    For Counter = 1 To NumBeeps
        Beep
    Next Counter
End Sub
```

Then the following statement, which you might type into a Visual Basic module, calls MultiBeep and passes the number three to it as an argument. To *pass* a value means to send it to a procedure.

```
MultiBeep 3
```

The procedure substitutes 3 for `NumBeeps` (the name that appears in the argument list), and the procedure causes your computer to beep three times.

How You Use Parentheses in a Procedure

When you define a **Sub** or **Function** procedure, you always follow the procedure name with parentheses, even if it has no arguments, as shown here. (Visual Basic adds these parentheses automatically, even if you don't type them.)

```
Sub StockSale ()
.
.
.
End Sub
```

However, when you call a **Sub** procedure, you do not usually use parentheses. For example, suppose the SetUpSheet procedure is modified so that it requires three arguments to set up your worksheet: one to specify a text heading, one to specify the number of transactions you plan to enter, and one to specify a date. The following revision of the StockSale procedure includes a call to the revised SetUpSheet procedure.

```
Sub StockSale ()
    SetUpSheet "Summary", 5, #Dec. 21, 1994#
.
.
.
End Sub
```

When you call a **Function** procedure that has arguments and you use its return value, you must enclose the arguments in parentheses. This is shown in the sixth

line of the following procedure, `Profit = StockSale(...)`. If you ignore the return value, however, you do not need parentheses. (This example uses the **Function** procedure version of the StockSale procedure shown earlier in this chapter.)

```
Sub DisplayProfit()
    OldPrice = InputBox("What was the purchase price?")
    Shares = InputBox("How many shares did you buy?")
    NewPrice = InputBox("What was the sale price?")
    'Calls the StockSale Function procedure
    Profit = StockSale(OldPrice, Shares, NewPrice)
    MsgBox "Your profit is " & Profit
End Sub
```

Note The same rules about parentheses that apply to **Function** procedures also apply to methods. You will learn more about methods in Chapter 5, "Working with Objects in Visual Basic."

You can enter the names of built-in functions or **Function** procedures directly in the argument list when you call procedures. For example, the following single line of code replaces the several lines of code in the preceding example. (The underscore is a line-continuation character that indicates the code should be typed all on one line.)

```
Profit = StockSale(InputBox("What was the purchase price?"), _
  InputBox("How many shares did you buy?"),
  InputBox("What was the sale price?"))
```

You can also simplify the calls to some procedures by using named arguments. With named arguments, you can pass specific arguments to a procedure in any order you want. For more information, see "Creating Simpler Statements with Named Arguments" in Chapter 6.

Organizing Procedures and Modules in Workbooks

So far in this chapter, procedures have called other procedures only when they are all within the same Visual Basic module—but they don't have to be. A Visual Basic module is just a sheet in your workbook, and you can add many modules to your workbook. You can also put many procedures in each module.

If one procedure calls a second procedure, Visual Basic first looks for the second procedure in the module containing the first one; if Visual Basic can't find the second procedure there, it looks in all the other Visual Basic modules in your workbook, and then in other workbooks to which a reference has been established. Visual Basic runs the second procedure as soon as it's found, no matter what module you keep it in. (For more information about establishing references between workbooks, see "Calling Procedures in Another Workbook" later in this chapter.)

Still, it's a good idea to organize procedures by placing similar ones in the same module. For example, you might place all your error-handling procedures in one module, all your math **Function** procedures in another, and all your automatic procedures in still another. (You'll learn about handling errors in Chapter 9, "Handling Errors and Error Values," and about creating automatic procedures in Chapter 13, "Creating Automatic Procedures and Add-in Applications.") Following is the procedure to insert a new module in your workbook.

▶ **To insert a new Visual Basic module into your workbook**

- From the Insert menu, choose Macro, and then choose Module.

After you insert a module, you can use the Sheet command on the Edit menu to give the new module an appropriate descriptive name.

How a Visual Basic Module Is Organized

A Visual Basic module usually contains several procedures. The order of the procedures doesn't matter. At the top of the module, you can place some lines above the first procedure. This top area is called the *Declarations* section of a module because it is where you declare some of your variables and constants; it is also where you can specify some Visual Basic options, and other advanced information. (You will learn about declaring variables and constants and specifying Visual Basic options in Chapter 6, "Working with Visual Basic Code in Procedures.")

You can place the items in the Declarations section in any order you want, but by convention most programmers specify options, then declare variables, and then declare constants, as shown in the following example.

```
'Top of module

'Declarations

'Specify Visual Basic options
Option Explicit
Option Compare Binary

'Declare variables
Dim OriginalPrice
Dim NumberOfShares
Dim SalePrice

'Declare constants
Public Const DOWJONES = 3500
Public Const S_P500 = 290

'End of Declarations section. Procedures follow.
```

```
Sub StockSale()
    SetUpSheet
    GetStockData
    InsertValues
End Sub

Sub SetUpSheet()
.
.
.
```

Calling Procedures in Specific Modules

As a general rule, you should give procedures unique names so that there is no ambiguity when you call them. But if you need to give two procedures in two different modules the same name, then you must include the module name when you call one of the procedures. Two procedures in the same module cannot have the same name.

To specify a module when you call a procedure, type the name of the module enclosed in square brackets, followed by a period and the name of the procedure. (The brackets are optional if the module name is one word and contains only letters and numbers.) For example, the following **Sub** procedure calls the SetUpSheet **Sub** procedure located in the Custom Formatting module.

```
Sub StockSale()
    [Custom Formatting].SetUpSheet
    GetStockData
    InsertValues
End Sub
```

The module specifier you type in your Visual Basic code is not automatically changed when you rename a module. If you change a module's name, you must also change all the corresponding module specifiers that precede the names of procedures; otherwise, Visual Basic is unable to find those procedures.

Note To rename a Visual Basic module while working within the module, you choose the Sheet command from the Edit menu and then choose Rename. When working in a Microsoft Excel worksheet, the Sheet command is located on the Format menu instead of the Edit menu.

Protecting Procedures from Calls by Procedures in Other Modules

You can prevent a procedure from being called outside the module containing it; to do this, you declare the procedure as *private*. This is helpful if a procedure serves no useful purpose outside its module or if it has the same name as a procedure that is called frequently from procedures in other modules.

To declare a procedure as private, precede the **Function** or **Sub** statement in the first line of the procedure with the **Private** keyword, as shown here.

```
Private Sub StockSale()
```

For more information about private procedures, search the online Visual Basic Reference for *Function* or *Sub* and read the description of the **Private** keyword.

Calling Procedures in Another Workbook

You can call procedures in another workbook only if you establish a reference from Visual Basic to that workbook. After you establish the reference, the workbook's modules (and thus all the procedures in those modules) are available to Visual Basic, and you call the procedures the same way you call procedures in your active workbook.

The ability to call procedures in another workbook is helpful if someone gives you a workbook or other add-in application that contains a set of procedures that you need to call but that might be inconvenient to copy into your workbook. For example, you can create a reference to the Analysis ToolPak add-in application to get access to its statistical, financial, and engineering functions. (For more information about the Analysis ToolPak, see Chapter 31, "Statistical Analysis of Data," in the *Microsoft Excel User's Guide.*)

▶ **To establish a reference to another workbook**

1. If you're working in a workbook, and the workbook to which you want to establish a reference (a second workbook) has not been saved, then switch to the second workbook and save it.
2. Switch back to the first workbook and, with a Visual Basic module active, choose References from the Tools menu.
3. In the Available References box, select the name of the second workbook.

   If the workbook name does not appear in the list, choose the Browse button to search for it.
4. Select the check box next to the name of the second workbook, if it is not already selected.
5. Choose the OK button.

You do not need to include the module specifier when you call the procedure, unless another procedure in your workbook or the referenced workbook has the same name.

You also use the preceding steps to create references to other desktop applications, such as Microsoft Word or Microsoft Project. This gives your procedures access to that application's objects, properties, methods, and so on. For more information about working with other applications, see Chapter 10, "Controlling and Communicating with Other Applications." For more information about objects, properties, and methods, see Chapter 5, "Working with Objects in Visual Basic."

Calling Procedures in a Specific Module in a Specific Workbook

After a workbook reference has been added, you can specify both a Visual Basic module name and a workbook name if you know exactly where the procedure is. To do this, type the name of the workbook enclosed in square brackets, followed by a period, and the name of the module enclosed in square brackets, followed by a period and the name of the procedure. (The brackets are optional if the workbook name or the module name is one word and contains only numbers and letters.)

For example, in Microsoft Excel for Windows, the following **Sub** procedure calls the SetUpSheet **Sub** procedure located in the Custom Formatting module in the VISUALS.XLS workbook.

```
Sub StockSale()
    [VISUALS.XLS].[Custom Formatting].SetUpSheet
    GetStockData
    InsertValues
End Sub
```

In Microsoft Excel for the Macintosh, the call might look like this:

```
Sub StockSale()
    [Visual Effects].[Custom Formatting].SetUpSheet
    GetStockData
    InsertValues
End Sub
```

If the name of the workbook changes, your Visual Basic code is not automatically updated, which is a disadvantage of specifying a workbook name. You usually need to specify the workbook name only if you have more than one procedure with the same name.

Protecting Modules from Calls by Procedures in Other Workbooks

You can prevent all procedures in a Visual Basic module from being called by procedures in another workbook. To do this, you declare the module as private. This is helpful if you want to distribute a workbook to other people but you don't want them calling procedures in certain modules. For example, you might have a module of procedures that maintain your workbook but that would be of no use to procedures in other workbooks.

To declare a module as private, include the **Option Private Module** statement anywhere in the module's Declarations section. For more information about the **Option Private Module** statement, search the online Visual Basic Reference for *Option Private.*

Tip Users of your workbook rarely need to see its private modules, so you might want to hide them completely. You can hide a Visual Basic module by choosing the Sheet Hide command from the Edit menu.

Restoring a Broken Link Between a Button or Graphic Object and a Visual Basic Module

In Chapter 1, "Automating Repeated Tasks," you learned how to assign procedures to buttons and other graphic objects. When you assign a procedure to a button or other graphic object, Microsoft Excel stores a link between the object and the assigned procedure. Because this link contains the name of the workbook where the procedure is stored, however, you can lose links if you rename the workbook outside of Microsoft Excel (for example, using the MS-DOS® Rename command). When you click an object that has an invalid link, Microsoft Excel displays a message that it could not find the procedure.

If you want to rename a workbook that has links to objects on other sheets in other workbooks, open all the affected workbooks, and then rename the first workbook using the Save As command on the File menu. If you have already lost the links to the workbook, you can use the following procedure to restore them.

▶ **To restore the link between a graphic object and a procedure**

1. Switch to the workbook that contains the object with the invalid link.
2. From the Edit menu, choose Links.
3. In the Source File box, select the previous name of the workbook.
4. Choose the Change Source button.
5. In the Change Links To box, type or select the new name of the workbook.

6. In Microsoft Excel for Windows, choose the OK button.

   In Microsoft Excel for the Macintosh, choose the Change button.

7. Choose the OK button.

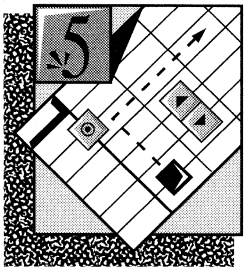
All the links to the previous workbook name—not just the one used by the object—are changed to refer to the new workbook. This procedure does not correct workbook names that you have typed in your code, such as `[MYBOOK.XLS]` or `[My Workbook]`. You must retype the new names in your code manually.

Note If you rename a procedure or a Visual Basic module containing procedures assigned to objects, you must choose the Assign Macro command from the Tools menu and manually reassign each procedure that you want assigned to an object. In other words, the Links command automatically takes care of changed workbook names but not changed module or procedure names.

In the next chapter, you will learn about the objects in Visual Basic and the Visual Basic code you use to work with those objects.

CHAPTER 5

Working with Objects in Visual Basic



With Visual Basic, you can make your procedures do more than simply replay the actions you record. Using Visual Basic, you can control almost anything you create with Microsoft Excel—even Microsoft Excel itself.

The way Visual Basic provides this control is by defining objects that you can manipulate in procedures. This chapter introduces you to objects and how you use them to automate the things you want to do in Microsoft Excel.

Contents

Introduction to Objects

An *object* is something you control with Visual Basic. You use objects to accomplish tasks in Microsoft Excel much the same way you use objects in everyday life. For example, a radio is an object you use to receive information and entertainment. In Microsoft Excel, a worksheet is an object you use to calculate and display information.

Each object in Microsoft Excel has characteristics that make that object useful to you, just as radios or other objects have. These characteristics are called *properties*—attributes that control the appearance or behavior of an object. For example, you could say a radio has a "Frequency" property that determines what station you hear.

In Microsoft Excel, worksheets have a **Visible** property you can use to determine whether a worksheet is visible. **Visible** is only one of the properties of a Worksheet object. You can set and return the values of properties for objects in your Visual Basic code to change how those objects behave. For more information about properties, see "Controlling Objects with Their Properties" later in this chapter.

In addition to properties, objects also have *methods*—actions that objects can do. You could say a radio has a "MakeLouder" method that increases the volume of the radio. Similarly, worksheets have a **Calculate** method that you can use to cause the entire worksheet to recalculate. **Calculate** is just one of the methods of a worksheet object. You use methods in your Visual Basic code to cause objects to do things you want them to do. For more information about methods, see "Performing Actions with Methods" later in this chapter.

Using Objects

When writing Visual Basic code, you first decide what objects will help you accomplish your task, and then you use those objects in the code. The code you write typically does one or more of three things:

- Changes the condition of an object by setting the value of one of the object's properties
- Examines the condition of an object by returning the value of one of the object's properties
- Causes the object to perform a task it can do by using a method of the object

For example, say you want to determine whether a range on a worksheet is empty. If it isn't empty, you want to clear the range. Once the range is empty, either because it was already empty or because you cleared it, you want to assign a formula to the range. The Visual Basic procedure you write would:

- Use a Range object to identify the range you want to manipulate.
- Return the value of the **Value** property of the range to determine whether the range is empty.
- Use the **Clear** method of the range to clear all the cells if the range isn't empty.
- Set the **Formula** property of the range to assign a formula to the range.

As you learn more about Visual Basic, you will discover that Microsoft Excel provides many objects you can use in code to automate your work, including:

- Workbooks
- Worksheets
- Ranges
- Charts

Although you might expect a cell on a worksheet to be an object, cells are not objects in Microsoft Excel. When you want to manipulate one or more cells with Visual Basic, you use the Range object; to manipulate a single cell, you use a Range object that contains that one cell. For more information about the Range object, see "Using Objects, Properties, and Methods in Code" later in this chapter.

Tip To learn more about how objects help automate tasks, use the Macro Recorder to record a task. You can look at the Visual Basic code produced by the Macro Recorder to find out how the recorded macro uses objects.

For more introductory information about object-oriented programming and objects in Microsoft Excel, search the online Visual Basic Reference for *Objects: understanding.*

Controlling Objects with Their Properties

The reason you write procedures in Visual Basic is to automate control of objects in Microsoft Excel. For example, you might want to change the width of a column in a worksheet. So you need your procedure to change the Range object. In Visual Basic, most objects have properties that govern their appearance and behavior.

You change objects by changing their properties. Think about a radio. What characteristics does a radio have that you change as you use it? One property of a radio is its volume. In Visual Basic, you would say a radio has a "Volume" property that you can change by changing its value. Assume you can set the volume of the radio from 0 to 10. If you could control a radio with Visual Basic, you might write code in a procedure that changes the value of the "Volume" property from 3 to 5 in order to make the radio play louder.

All the objects in Microsoft Excel have properties. For example, the following table briefly describes a few of the properties of the Range object.

Property	Description
Column	Returns the first column of the first area in the range
Formula	Returns the range's formula, in A1-style notation
Height	Returns the height of a range, in points (1/72 inch)
WrapText	Determines whether text wraps inside the range

You can change the values of these properties to change the appearance or behavior of a Range object. For example, assume you want to widen all columns in a range to better display the information in the range. Part of your procedure might change the value of the **Width** property of the range accordingly. If some of the cells in the range contain a sentence or two, another part of the procedure might change the value of the **WrapText** property so the cells display multiple lines of text.

So part of what you do when you use objects in your procedures is set the values of their properties. You can also return the value of a property. For example, say you want to check a computed value in a cell to decide whether to run additional code. Your procedure can return the **Value** property of that cell and use the result as the basis for deciding whether to run the additional code.

Referring to Properties

Before you can set or use properties, you need to know how to refer to a property in Visual Basic code. When you refer to a property, the object whose property you want to set or return comes first, followed by a period and then the name of the property. The following is an example of how to refer to the **ColumnWidth** property of a Range object. The **Cells** method returns a Range object.

```
Cells.ColumnWidth = 16
```

You separate the object and its property with a period. The period tells Visual Basic where the object ends and the name of property begins. (For more information about the **Cells** method and the Range object, see "Using Objects, Properties, and Methods in Code" later in this chapter.)

Using Property Values in Procedures

When you use properties in Visual Basic, you do one of two things:

- Set the value of a property
- Return the value of a property

With most properties, you can either set or return their values according to the needs of your procedure. Properties whose value you can either set or return are called read-write properties. With other properties, you can only return their values. These properties are called read-only properties. For more information about whether a particular property is read-write or read-only, search the online Visual Basic Reference for the name of the property.

Types of Property Values

In most cases, the value of a property is one of three types:

- A numeric value, such as **14** for the setting of the **RowHeight** property of a cell
- A character string, such as "Annual Totals" for the setting of the **Value** property of a cell
- A **True** or **False** value, such as **True** for the setting of the **Bold** property of a Font object

Knowing the type of value that applies to a given property can help you make sure that you don't try to set a property to a value it cannot accept. For example, when setting the properties of a Font object, you need to remember that the **Name** property requires a character string to identify the font, while the **Bold** property requires either **True** or **False**.

There are other types of property values. For more information about the type of values you can use for a given property, search the online Visual Basic Reference for the name of that property.

Setting a Property Value

You set the value of a property when you want to change the appearance or behavior of an object. For example, you change the **Value** property of a cell to change the content of that cell on the worksheet. In this case, you change how that cell appears on the worksheet in addition to changing the information it contains. Another example is setting the value of the **WrapText** property of a cell. In this case, the behavior of the cell switches between wrapping and not wrapping text.

To set the value of a property, use the following syntax:

object.property = expression

where *object* is a reference to an object (possibly including its containers), *property* is the name of a property of the object, and *expression* is the value to which you want to set the property. For more information about containers, see "Objects as Containers for Other Objects" later in this chapter.

The following statements demonstrate how you set properties.

```
ActiveCell.RowHeight = 14
ActiveCell.Value = "Annual Totals"
ActiveCell.Locked = False
```

Returning a Property Value

You return the value of a property when you want to determine the condition of an object before your procedure performs additional actions (such as assigning the value of the property to another object). For example, you can return the **Value** property of a cell to determine the contents of the cell before running code that might change the value of the cell.

In most cases, to return the value of a property, you use the following syntax:

variable = object.property

where *variable* stands for a variable or another property in which you store the returned property value, *object* is a reference to an object (possibly including its containers), and *property* is the name of a property of the object.

You can also return a property value as part of a more complex expression without assigning the property to a variable. For example, you might calculate the position of the right edge of a Button object by adding the value of the **Left** property to the value of the **Width** property.

```
If MyButton.Left + MyButton.Width > 350 Then
    MyButton.Width = MyButton.Width / 2
End If
```

In this example, the first statement returns the **Left** and **Width** properties of a Button object called MyButton. This statement adds the returned property values and uses the result to decide whether to execute the second statement.

You can also return the value of a property in order to set the value of another property. For example, you might want to make the **Formula** property of a certain cell match the **Formula** property of another cell.

```
ActiveSheet.Cells(1, 1).Formula = ActiveSheet.Cells(1, 3).Formula
```

This statement returns the value of the **Formula** property of cell C1 to set the value of the **Formula** property of cell A1 to the same value.

Common Properties

Some properties are common to many objects in Microsoft Excel or are properties you use frequently. These are important properties to learn about as you start using Visual Basic. The following table lists some of these properties and briefly describes how they affect objects in Microsoft Excel.

Property	Description
ActiveCell	The active cell of the active window.
ActiveSheet	The active sheet of the active workbook.
ActiveWorkbook	The active workbook in Microsoft Excel.
Bold or **Italic**	The type style of the text displayed by a Font object, either bold or italic.
Column or **Row**	A number that identifies the first column or row of the first area in a range.
ColumnWidth	The width of all columns in the specified range.
Height or **Width**	The height or width of an object, in points. This property applies to many different objects in Microsoft Excel.
RowHeight	The height of all of the rows in the specified range, in points.
Selection	The object currently selected in the active window of the Application object, or the current selection in the given Window object.
Value	The value of a cell.

Performing Actions with Methods

In addition to properties, objects have methods. For example, a telephone is able to ring when it receives a call. You might say that telephones have a "Ring" method. In Microsoft Excel, a range of cells can clear their contents. So Range objects have a **Clear** method. Some methods identify objects you can use in your code. For example, you can use the **Range** method to identify a range you want to use in a procedure.

Methods are a part of objects just as properties are. The difference between methods and properties is that properties have values which you set or return, while methods are actions you want an object to perform. Also, most properties take a single value, whereas methods can take one or more arguments.

Compare the table of Range object properties in "Controlling Objects with Their Properties" earlier in this chapter with the following table, which briefly describes some methods of a Range object.

Method	Description
Calculate	Calculates a specified range of cells in a sheet
Clear	Clears the contents of the entire range
Copy	Copies the range to the Clipboard
Justify	Rearranges the text in a range so that it fills the range evenly
Table	Creates a data table based on input values and formulas that you define on a worksheet

Methods can affect the values of properties. In the radio analogy, for example, the "MakeLouder" method changes the "Volume" property. Similarly, in Microsoft Excel, each cell in a range has a **Value** property. The Range object that contains the cell has a **Clear** method. When a procedure uses the **Clear** method on a Range object, the **Value** property of all the cells in the range changes.

Using Methods in Code

When you use the method of an object in your procedure, how you write the code depends on whether the method takes arguments. When a method does not take arguments, you write the code using the following syntax:

object.method

where *object* is the name of the object (possibly including the names of its containers) and *method* is the name of the method. For example, the **Justify** method of a Range object doesn't take arguments. To justify the cells in a Range object named Price, you write:

```
Price.Justify
```

When a method does take arguments, however, the way you write your code depends on whether you want to save the value returned by the method. If you don't want to save the return value of a method, the arguments appear without parentheses as shown here:

object.method arguments

where *object* and *method* represent the name of the object and its method. If the method takes more than one argument, you separate the arguments with a list separator and a space. For example, the **Table** method of a Range object can take two arguments. To create a data table from a range named Price, you write:

```
Range("Price").Table Cells(2, 1), Cells(3, 1)
```

The return value of the **Table** method is not saved, so the two arguments passed to the **Table** method appear without parentheses around them.

Note The decimal separator, list separator, currency symbol, and date formats that you use when you're writing or editing code are determined by the Language/Country settings on the Module General tab of the Options dialog box (Tools menu). The decimal separator, list separator, currency symbol, and date formats that users see when they run your code are determined by the country setting of their operating system.

The code examples in this book assume you use U.S. country settings. For more information about writing applications for use in other countries, see Appendix A, "Writing Code for International Use."

If you do save the return value of a method, however, you must enclose the arguments in parentheses. For example, the **CheckSpelling** method of a worksheet returns a value (**True** or **False**) that indicates whether the text is spelled correctly.

```
ThisWord = ActiveSheet.CheckSpelling(ignoreUppercase:=True)
```

The return value of the **CheckSpelling** method is saved in the variable `ThisWord`. Therefore, the argument passed to the method must appear inside parentheses.

Using Named Arguments

Most methods can receive arguments in either of these forms:

- Conventional
- Named

In the previous example referring to the **Table** method, the arguments are passed as conventional arguments. *Conventional arguments* appear in a defined order, and Visual Basic interprets their values according to the order in which you pass them.

However, you can also use *named arguments*—arguments identified by name rather than by position in the syntax. For example, to use the **Table** method with named arguments, you can write:

```
Price.Table ColumnInput := Cells(3, 1), RowInput := Cells(2, 1)
```

Named arguments are often easier to understand when reading code and are easier to use. For more information about the difference between conventional arguments and named arguments, see "Creating Simpler Statements with Named Arguments" in Chapter 6.

Working with a Collection of Objects

In Microsoft Excel, a *collection* is a set of related objects. For example, the Worksheets collection contains all the worksheets in a given workbook. The Toolbars collection contains all the toolbars in the Microsoft Excel Application object. Each object within a collection is called an *element* of that collection.

Collections are objects themselves. Because collections are objects, they have their own properties and methods. You can use these properties and methods to control individual elements in the collection as well as all objects in the collection.

Methods That Return Objects

Some objects have special methods that return a reference to another object, usually a collection. For example, the Workbook object has a **Worksheets** method you can use to return a reference to the entire collection of worksheets in the workbook or to an individual worksheet in the Worksheets collection.

Note An example of a method that returns an object that isn't a collection is the **Cells** method, which returns a Range object. For more information about the **Cells** method, see "Using Objects, Properties, and Methods in Code" later in this chapter.

Returning an Individual Object in a Collection

When you need to work with one element of a collection, you must specify which element you want. For example, assume you want to identify a single worksheet in a workbook. You can use the **Worksheets** method to specify one sheet in the collection.

```
Worksheets("MySheet")
```

As you can see, the **Worksheets** method requires an argument to refer to a specific worksheet. In this case, the argument is the name of the worksheet MySheet. You can also use a number to identify an element of a collection.

```
Worksheets(3)
```

When you use a number to identify an element in the collection, that number is called an *index*. The first index in a collection is 1. In most cases, the index is a single number.

Returning All the Objects in a Collection

In some procedures, you want to make changes that affect all the objects in a collection. For example, to delete all the drawing objects on Sheet1, you might use the following statement.

```
Worksheets("Sheet1").DrawingObjects.Delete
```

This statement uses the **Worksheets** method to return a reference to a single worksheet. However, because no index is specified for the DrawingObjects collection, all the elements in the collection are deleted. (When you use the **Worksheets** method without specifying an index, the returned reference is to the entire collection instead of a single element.)

You can also write a loop that contains code that runs on every object in the collection. For example, to close every workbook in the Workbooks collection, you can use the following code.

```
For Each Book In Workbooks
    Book.Close
Next
```

In this code, `Book` is a variable used to represent each workbook in the Workbooks collection. The code inside the loop uses the **Close** method on the object specified by the variable to close each workbook in the collection.

For more information about writing loops, see "Running the Same Code Several Times (Looping)" in Chapter 7.

Creating a New Object in a Collection

When you work with collections, you typically work with the objects already in the collection. However, you might need to add new objects to a collection. You can use the **Add** method to add a new object to most collections in Microsoft Excel.

The **Add** method can take the initial property settings of the new object as its arguments. For example, to add a new scenario called Likely to the active worksheet, you can use the following statement.

```
ActiveSheet.Scenarios.Add _
    name:="Likely", _
    changingCells:=Range("A1:C5"), _
    comment:="This year's likely case"
```

This statement sets the initial values of the **Name** and **Comment** properties of the new scenario, as well as the scenario's values.

For more information about scenarios, search the online Visual Basic Reference for *Scenarios.*

Common Collections

In Microsoft Excel, you encounter and use some collections more frequently than others. The following table lists and briefly describes the most commonly used collections in Microsoft Excel.

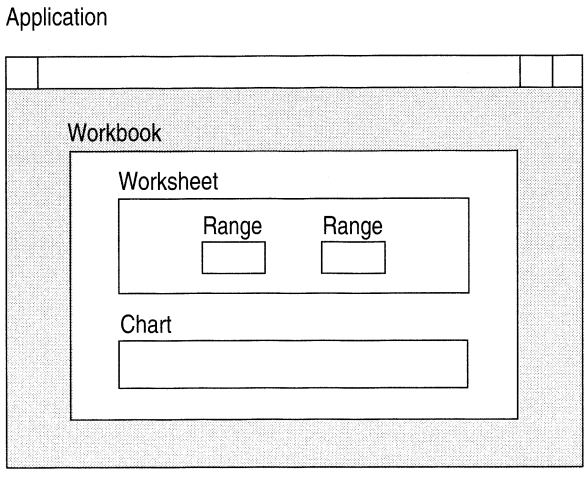
Collection	Description
Sheets	Contains all the sheet objects in a workbook, including Worksheet, Chart, Visual Basic Module, and Dialog Sheet objects.
Worksheets	Contains all the Worksheet objects in a workbook.
Charts	Contains all the Chart sheets in a workbook. (Embedded chart objects are contained in the ChartObjects collection.)
Workbooks	Contains all the open workbooks in Microsoft Excel.

For more information about these and other collections in Microsoft Excel, search the online Visual Basic Reference for *Objects.*

Objects as Containers for Other Objects

Some objects in Microsoft Excel contain other objects. For example, a workbook contains one or more worksheets. A worksheet contains rows, columns, and ranges. The outermost container for objects in Microsoft Excel is the application itself (called the Application object). All other objects you use are contained within the application. The application contains global objects, such as the menu bar, in addition to individual workbooks.

The relationship of objects to their containers is shown in the following illustration. The application acts as a container for all the open workbooks. The workbook is a container that includes a worksheet and a chart.



Worksheets act as containers for individual ranges, rows, and columns. In fact, worksheets provide a good example of how an object can be in more than one container at the same time. Think about a range of cells on a worksheet. The worksheet where that range resides is also a container for those cells, and the same range might exist inside a row or column, either of which is also a container.

Using Containers in Your Procedures

The advantage of having objects act as containers for other objects is that you can refer to the container in your procedure to make it clear which object you want to use. For example, you could have two different worksheets in a workbook, each of which has a cell identified by Cells(1,1). Say you want to write a procedure that puts information in the first cell of one worksheet, but not the other. You can specify exactly which cell you want to use by referring to the worksheet containing the range.

```
ActiveSheet.Cells(1, 1).Value = 42
```

In this sample code, the reference to the object (a range) includes a reference to the worksheet that contains the range. Including the reference to the container makes it clear that you want to change the value of the first cell on the active worksheet rather than on another worksheet in the workbook.

When you use a reference to an object that includes references to one or more of its containers, Visual Basic has to distinguish between each piece of the reference. For example, in the previous code sample, you need to make clear that `ActiveSheet` is a reference to a worksheet and `Cells(1, 1)` is a reference to a Range object within the worksheet.

To identify where one reference ends and the next one begins, separate them with a period. If you don't separate references to objects from references to their containers, Visual Basic cannot read your code accurately. For instance, without the period between `ActiveSheet` and `Cells(1, 1)`, the previous example looks like this:

```
ActiveSheetCells(1, 1).Value = 42
```

Visual Basic incorrectly interprets `ActiveSheetCells(1, 1)` as an element of a variable array; because variable arrays don't have a **Value** property, this statement creates an error.

Using the Object Browser

While writing your Visual Basic procedures, you might want to see what objects are available in the active workbook, including the names of procedures you have already written. You can look at this information using the Object Browser.

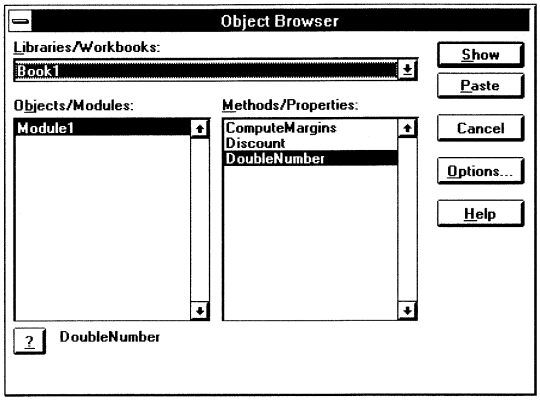
The Object Browser performs three important functions:

- Gives you a quick way to navigate in your code.
- Shows you what objects are available for your procedures, including the properties and methods of those objects.
- Gives you a way to paste code templates into a module. You can edit the pasted code to make writing procedures faster and less subject to typing errors.

▶ **To display the Object Browser**

- From the View menu, choose Object Browser.

  You can also click the Object Browser button on the Visual Basic toolbar.



What Appears in the Object Browser

The Object Browser dialog box displays either of two sets of information:

- The names of all the objects in a library, in addition to all the properties and methods of each object
- The names of all the Visual Basic modules in a workbook, in addition to the names of all the procedures in each module

The Object Browser contains a box that lists the names of all the libraries and workbooks referenced by the workbook containing the module you are creating. A *library* is a file or a part of an application that provides Visual Basic with information about objects. Libraries and workbooks identify the objects you can use in your procedures. For more information about references to libraries and workbooks, see "Adding References to Applications and Workbooks" in Chapter 10.

The Libraries/Workbooks list includes:

- Microsoft Excel, which provides a library of objects specific to Microsoft Excel.
- VBA (Visual Basic for Applications), which provides functions and other language features you can use in Microsoft Excel.
- The open workbook, which contains your code.
- Any other libraries referenced by the code in your workbook.

Below the Libraries/Workbooks box is the Objects/Modules box. The contents of the Objects/Modules box depend on whether the Libraries/Workbooks box shows the name of an object library or a workbook, as described in the following table.

If the Libraries/Workbooks box shows	Then the Objects/Modules box shows
The name of an object library	The available objects in the library, or categories of available functions
The name of a workbook	The names of modules in the workbook

To the right of the Objects/Modules box is the Methods/Properties box. The contents of the Methods/Properties box depend on whether the Objects/Modules box shows the objects in a library or the modules in a workbook, as described in the following table.

If the Objects/Modules box shows	Then the Methods/Properties box shows
The objects in an object library	The properties and methods of each object in the library
The modules in a workbook	The names of the procedures in each module, including property procedures

For more information about properties and methods, see "Controlling Objects with Their Properties" and "Performing Actions with Methods" earlier in this chapter. For more information about property procedures, see "Learning More About Visual Basic Programming" in Chapter 6.

Moving Between Procedures

You can use the Object Browser to move quickly from one procedure in a workbook to another.

▶ **To move to a procedure in a workbook**

1. Switch to a Visual Basic module.
2. From the View menu, choose Object Browser.
3. In the Objects/Modules box, select the name of the module containing the procedure you want.
4. In the Methods/Properties box, select the name of the procedure.
5. Choose the Show button.

The Object Browser moves you to the module containing the procedure you selected. The insertion point appears beside the line immediately following the line containing the procedure's **Sub** or **Function** statement.

Browsing Objects

You can also use the Object Browser to learn:

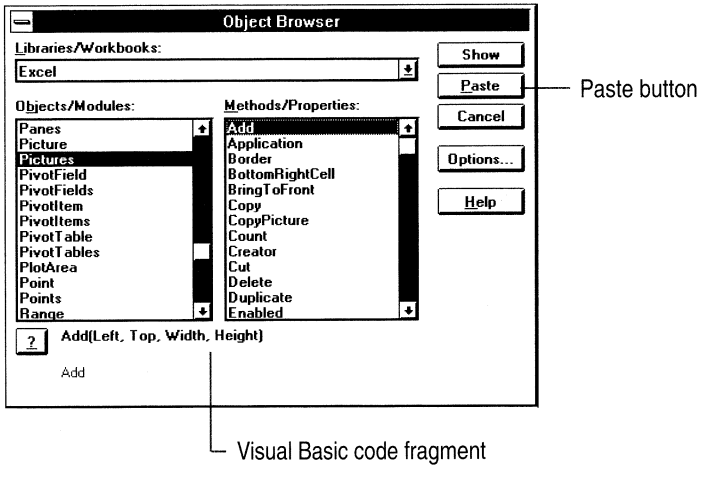
- The objects available in the libraries referenced by your workbook.
- The properties and methods of those objects.

▶ **To find the properties and methods of objects**

1. From the View menu, choose Object Browser.
2. In the Libraries/Workbooks box, select the name of the object library that you want to browse or that contains the object you want to use.
3. In the Objects/Modules box, select the name of the object for which you want to find properties and methods or that you want to use.

Pasting Code

Once you use the Object Browser to find an object with a property or method you want to use, you can paste a fragment of Visual Basic code from that property or method into your procedure. You can then edit the code fragment to fit the needs of your procedure. The Object Browser pastes the code fragment at the insertion point in the active module.



The code fragment includes syntax for named arguments when appropriate. For more information about named arguments, see "Creating Simpler Statements with Named Arguments" in Chapter 6. After you paste the code fragment into your procedure, you can delete any arguments you don't need and add the appropriate values for the arguments you decide to use.

▶ **To paste a code fragment into a procedure**

1. In a Visual Basic module, place the insertion point at the location in the code where you want to paste the code fragment.
2. From the View menu, choose Object Browser.
3. In the appropriate boxes, select the library and the object whose property or method you want to use.
4. In the Methods/Properties box, select the property or method you want to paste into the procedure.
5. Choose the Paste button.

Performing Multiple Actions on an Object

Procedures often need to perform several different actions on the same object. For example, you might need to set several properties for the same cell. One way to do this is to use several statements.

```
ActiveSheet.Cells(1, 1).Formula = "=SIN(180)"
ActiveSheet.Cells(1, 1).Font.Name = "Arial"
ActiveSheet.Cells(1, 1).Font.Bold = True
ActiveSheet.Cells(1, 1).Font.Size = 8
```

Notice that all these statements use the same object reference. You can make this code easier to enter, easier to read, and actually more efficient to run by using the **With** statement.

```
With ActiveSheet.Cells(1, 1)
    .Formula = "=SIN(180)"
    .Font.Name = "Arial"
    .Font.Bold = True
    .Font.Size = 8
End With
```

You can also nest **With** statements. For example, you can rewrite the preceding code example using one **With** statement nested inside another.

```
With ActiveSheet.Cells(1, 1)
    .Formula = "=SIN(180)"
    With .Font
        .Name = "Arial"
        .Bold = True
        .Size = 8
    End With
End With
```

For more information about nesting Visual Basic code, see "Placing One Control Structure Inside Another (Nesting)" in Chapter 7.

The object reference following the **With** statement can be a reference to a single object or a collection. Here is an example of setting the font properties of all the text boxes on the active sheet.

```
With ActiveSheet.TextBoxes
    .Font.Name = "Arial"
    .Font.Size = 8
End With
```

Using Objects, Properties, and Methods in Code

This section shows you how objects, properties, and methods actually work together in Visual Basic code—beginning with one of the most commonly used objects, the Range object. The code examples that follow illustrate some of the ways you can manipulate objects in Visual Basic and fine-tune your control of your application.

The Range Object

A Range object can consist of:

- A cell.
- A row or column.
- One or more selections of cells.
- A 3-D range.

Identifying a Range Object with the Cells Method

The most common way to identify a Range object is to use the **Cells** method. The **Cells** method returns a Range object, which is a collection of cells. Because cells are organized into rows and columns on a worksheet, it is often helpful to use two indexes to specify a cell. This way you can specify the row and column intersection of a cell by passing one index for the row and a second index for the column.

For example, you can identify cell A1 on the active worksheet with the following code (the worksheet must be active when you run the code).

```
Cells(1, 1).Value = 24              'Set the value of cell A1 to 24
```

The first number identifies the row of the cell, and the second identifies the column. In the case of the **Cells** method, the index for the column can be the letter that appears at the top of the column instead of the column number. To identify cell A1 using a letter for the column index, you write:

```
Cells(1, "A").Value = 24            'Set the value of cell A1 to 24
```

Although you can use a letter to identify a column index in the **Cells** method, this practice doesn't always result in the most useful code. For example, by using only numeric row and column indexes, you can write the following procedure. (A worksheet must be active when you run this procedure.)

```
Sub SetUpTable()
    For TheYear = 1 To 5
        Cells(1, TheYear + 1).Value = 1990 + TheYear
    Next TheYear
    For TheQuarter = 1 To 4
        Cells(TheQuarter + 1, 1).Value = "Q" & TheQuarter
    Next TheQuarter
End Sub
```

Although you can use Visual Basic string functions in this procedure to manipulate A1-style references, it's much easier (and much better programming practice) to use numeric row and column indexes.

For very simple procedures, Visual Basic provides a shortcut syntax for referring to cells: Enclose the absolute, A1-style reference in square brackets. For example, `[A1].Value = 24` sets the value of cell A1 to 24. For more information about the **Cells** method, search the online Visual Basic Reference for *Cells method.*

Using the Range Method

You can use the **Range** method to return a rectangular range of cells. The following example fills the range A1:H8 with the string "Test."

```
Sub FillTheRange()
    Range(Cells(1, 1), Cells(8, 8)).Value = "Test"
End Sub
```

You can also use the **Range** method to return a named range, as shown in the following example.

```
Sub ClearDatabaseCriteria()
    Range("Criteria").ClearContents
End Sub
```

The Range method provides additional shortcuts for referring to ranges of cells in very simple procedures. For example, `Range("A1") = 24` sets the value of cell A1 to 24. For more information about the **Range** method, search the online Visual Basic Reference for *Range method.*

Multiple Selections

You can use the **Union** method and **Range** method to return multiple selections. The following example creates an object called MyMultipleSelection, defines it as the multiple selection of A1:B2 and C3:D4, and then selects it.

```
Sub ShowMultipleSelection()
    Dim r1, r2, MyMultipleSelection As Range
    Set r1 = Range(Cells(1, 1), Cells(2, 2))
    Set r2 = Range(Cells(3, 3), Cells(4, 4))
    Set MyMultipleSelection = Union(r1, r2)
    MyMultipleSelection.Select
End Sub
```

If you work with multiple selections, the **Areas** method is very useful. It divides a multiple selection into individual Range objects and then returns them as a collection. You can use the **Count** property on the returned collection to check for a multiple selection, as shown in the following example.

```
Sub NoMultipleSelections()
    NumberOfSelectedAreas = Selection.Areas.Count
    If NumberOfSelectedAreas > 1 Then
        MsgBox "Command not available on multiple selections"
    End If
End Sub
```

Examples of Objects, Properties, and Methods in Code

This section presents four procedures that demonstrate how objects, properties, and methods work together in Visual Basic code. You might not be able to understand all the code yet, but read it to see objects in the context in which they're actually manipulated.

These examples use many of the objects, properties, and methods described earlier in this chapter and introduce some new ones. For a description of any Visual Basic keywords that interest you, search the online Visual Basic Reference for the name of that keyword. (If you're typing these examples in a Visual Basic module, you can also select the keyword and press the F1 key in Microsoft Excel for Windows or COMMAND+SLASH in Microsoft Excel for the Macintosh.)

Applying a Fancy Format to a Range

This procedure takes a square selection of cells and makes a regular pattern of rows and columns in the range bold. Look for the following elements of the Visual Basic programming language:

- The **ActiveCell** property.
- The **CurrentRegion** property. (The current region is an area bounded by blank rows and columns.)
- The **Resize** method. (This method resizes the current selection.)
- The Font object.

- The **Color** property.
- The **RGB** function. (This function returns a number corresponding to a color.)

```
Sub FancyFormat()
    Blue = RGB(0, 0, 255)                    'The color blue
    Black = RGB(0, 0, 0)                     'The color black
    'Makes alternating rows and columns bold.
    Set SquareRange = ActiveCell.CurrentRegion
    RowNum = SquareRange.Rows.Count
    'Loop through rows and columns, alternating bold formatting in a
    'fancy pattern.
    BoldFlag = True
    For I = RowNum To 1 Step -1
        Set SquareRange = SquareRange.Resize(I, I)
        SquareRange.Font.Bold = BoldFlag
        If BoldFlag = False Then
            SquareRange.Font.Color = Blue
        Else
            SquareRange.Font.Color = Black
        End If
        BoldFlag = Not (BoldFlag)
    Next I
End Sub
```

The following illustration shows the selected range with the bold formatting applied.

	A	B	C	D	E
1	**Fancy**	Fancy	**Fancy**	Fancy	**Fancy**
2	Fancy	Fancy	**Fancy**	Fancy	**Fancy**
3	**Fancy**	**Fancy**	**Fancy**	Fancy	**Fancy**
4	Fancy	Fancy	Fancy	Fancy	**Fancy**
5	**Fancy**	**Fancy**	**Fancy**	**Fancy**	**Fancy**

Creating a Chart Based on Worksheet Data

This procedure creates a 3-D column chart based on the information in cells A1:D6, as shown in the following illustration.

	A	B	C	D
1		Widgets	Cogs	Sprockets
2	1989	10	12	15
3	1990	11	14	18
4	1991	12	16	21
5	1992	13	18	24
6	1993	14	20	27

In the procedure, look for the following elements of the Visual Basic programming language:

- The **Range** method.
- The **Select** method.

- The **Add** method.
- The **Charts** method.
- The **Set** statement. (This statement assigns an object to a variable.)

```
Sub ChartDemo()
    'Declare variables
    Dim ChartRange As Object
    Dim NewChart As Object
    'Define the range with values to be used in the chart
    Set ChartRange = Worksheets(1).Range("A1:D6")
    ChartRange.Select
    'Add a chart sheet (based on the chart range)
    'Assign the new chart to a variable.
    Set NewChart = Charts.Add
    NewChart.Activate
    'Select the chart type and subtype
    With NewChart
        .Type = xl3DColumn
        .SubType = xlNormal
    End With
End Sub
```

Finding and Printing Workbook Files

This procedure displays a simple message box and asks the user to enter a drive and directory, such as C:\TEMP\. The procedure then searches the directory for workbook files (*.XLS), opens each file, and prints it. In the following code, look for these elements of the Visual Basic programming language:

- The **InputBox** function.
- The **Dir** function. (This function returns a list of filenames.)
- The **Do...Loop**. (This statement repeats the same code several times while a particular condition is **True**.)
- The **Open** method.
- The **Print** method.

```
Sub PrintBooks()
    'Prompt user for particular directory to search for workbooks and
    'then print the workbooks.
    DirName = InputBox("Enter the directory to search:", "Print Books")
    NextBook = Dir(DirName & "*.xls")
```

```
    'Loop condition is True as long as a workbook is found in the
    'specified directory.
    Do While NextBook <> ""
        MsgBox NextBook & " is being opened"
        'Open the workbook
        Workbooks.Open DirName & NextBook
        'Print the workbook
        ActiveWorkbook.PrintOut
        'Close the workbook
        MsgBox NextBook & " is being closed"
        ActiveWorkbook.Close
        'To get any additional filenames that match path, call
        'Dir again with no arguments.
        NextBook = Dir()
    Loop
End Sub
```

Verifying Data on a Worksheet

The following procedure verifies data entered on a worksheet that contains an inventory of parts. The first column in the worksheet contains a quantity, which must be a number. The second contains a **True** or **False** value, indicting whether the part has been ordered. The third column contains a part number, the first five characters of which must be numbers and the remaining characters of which can be either numbers or letters. The sample data might look like the following illustration.

	A	B	C
1	Quantity	On Order?	Part Number
2	100	TRUE	12345-3XCD
3	200	TRUE	14365-75HD
4	250	TRUE	18675-67GR
5	350	FALSE	53985-RY57
6	800	FALSE	43785-T6H6
7	210	TRUE	54980-3D5T
8	440	FALSE	56789-9KJ7

In the code example, look for the following elements of the Visual Basic programming language:

- The **Set** statement.
- The **CurrentRegion** property.
- The **Range** method.
- The **Rows** method.
- The **Count** property.
- The **For Each...Next** statement. (This statement repeats the same code several times, once for each element in a collection.)

- The **IsNumeric** function. (This function returns **True** if its argument is a number.)
- The **MsgBox** statement. (This statement displays a message on the screen.)

This procedure assumes that a title row exists, and it checks each column to see whether every value in the column is of the same type. The three columns must contain (in this order) numbers, Booleans, and serial numbers, with the first five characters required to be numbers.

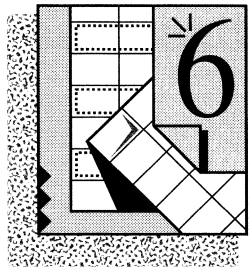
```
Sub TypeCheckRange()
    'Set range variable to current range area
    Set RangeToCheck = ActiveCell.CurrentRegion
    'Set new range below the title row
    Set NewRange = Range(RangeToCheck.Cells(2, 1), _
        RangeToCheck.Cells(RangeToCheck.Rows.Count, _
            RangeToCheck.Columns.Count))
    'Verify each cell in the Quantity column
    For Each Cell In NewRange.Columns(1).Cells
        If Not (IsNumeric(cell.Value)) Then
            MsgBox "You must enter a number"
        End If
    Next
    'Verify each cell in the On Order column
    For Each Cell In NewRange.Columns(2).Cells
        If Not (cell.Value = True Or cell.Value = False) Then
            MsgBox "You must enter a Boolean value"
        End If
    Next
    'Verify each cell in the Part Number column
    For Each Cell In NewRange.Columns(3).Cells
        If Not (IsNumeric(Left(cell.Value, 5))) Then
            MsgBox "The first 5 characters must be numeric"
        End If
    Next
End Sub
```

Important Many of the examples throughout this chapter assume a worksheet is active before you run them. If you experience problems when trying to run your code from a Visual Basic module, either switch to a worksheet and then run the code, or insert a line similar to the following, above the line causing the error:

```
Worksheets("Sheet1").Activate
```

CHAPTER 6

Working with Visual Basic Code in Procedures



In earlier chapters, you learned about procedures, objects, properties, and methods. In this chapter and Chapter 7, "Controlling How Your Code Runs," you will learn about important Visual Basic terms and concepts, what makes your code work, and how to use code to direct the flow of your procedures.

Contents

This chapter is organized in order of complexity, with the simplest information about Visual Basic arguments, variables, and data types presented first. The rest of the chapter expands on the topics presented in the first part and then examines other advanced features.

Understanding Visual Basic Code

Procedures are composed of Visual Basic code. Following is a list of terms and concepts that describe Visual Basic code. Understanding these terms will help you as you learn about Visual Basic in this chapter.

- The *Visual Basic code* in a procedure tells it what to do. The fundamental unit of Visual Basic code is the statement.
- A *statement* is a syntactically complete unit of code that expresses one action, declaration, or definition. Each of the following lines of Visual Basic code is a statement.

  ```
  Cells(1, 1).Value = 4 'Sets the value property of cell A1 to 4
  MsgBox "You cannot delete this item." 'Displays a message box
  Profit = Revenue - Cost 'Assignment statement
  ```

  Sometimes a statement contains other statements or functions. For example, the **If...Then...Else** statement always contains other statements within it.

  ```
  If TotalSalePrice <= 15000 Then
      Commission = 25 + 0.03 * SharesSold
  Else
      Commission = 25 + 0.03 * (0.9 * SharesSold)
  End If
  ```

  Because statements such as **If...Then...Else** contain several lines of code, they are sometimes called *statement blocks.*
- A *function* returns a value. For example, the built-in **Hour** function returns the hour of the day, and the **UCase** function returns uppercase text. A ***Function procedure*** is a type of function that you create.
- Some keywords in Visual Basic can be either functions or statements, depending on how you use them. For example, there is a **Date** statement and a **Date** function.

  ```
  Date = #02-28-1994# 'The Date statement sets the system date.
  TodaysDate = Date   'The Date function returns the current date.
  ```

Tip If you forget what a Visual Basic term means, search the online Visual Basic Reference for *Glossary.*

Common Types of Visual Basic Statements

Many of the statements in your Visual Basic code appear in one of the following forms:

- Assigning values to variables (referred to as an assignment statement)

  variable = *expression*

  ```
  UsersAge = InputBox("What is your age?")
  ```

- Executing built-in statements (syntax varies from statement to statement)

  statement argumentlist

  ```
  Name "OLDFILE.TXT" As "NEWFILE.TXT" 'Renames a file
  ```

- Controlling how your code runs based on conditions

  ```
  Do
      AnyNumber = InputBox("Enter a number")
  Loop Until IsNumeric(anyNumber)
  MsgBox "The square root is: " & Sqr(anyNumber)
  ```

- Setting properties of objects

  object.Property = *expression*

  ```
  ActiveCell.Font.Bold = True
  ```

- Executing methods of objects

  object.Method argumentlist

  ```
  Application.CheckSpelling word:="norepinephrine"
  ```

How to Use Parentheses with Arguments

Sub procedures, built-in statements, and some methods do not return a value, so when you call them you do not enclose their arguments in parentheses. **Function** procedures, built-in functions, and some methods do return a value, so when you call them you enclose their arguments in parentheses. If you're ignoring the return value, or if you're not passing arguments at all, do not use parentheses. These guidelines apply whether or not you're using named arguments. For more information about named arguments, see "Creating Simpler Statements with Named Arguments" later in this chapter.

```
FormatMySheet 15, 25                    'Calls a Sub procedure.

FormatMySheet NumRows:=15, RowSize:=25  'Calls a Sub procedure using
                                        'named arguments.

X = CalculateProfit(100, 50, 1000)      'Calls a Function procedure;
                                        'stores the return value.
```

```
CalculateProfit 100, 50, 100      'Calls a Function procedure; ignores
                                  'the return value.

'Executes a method; stores the return value.
SpelledRight = Application.CheckSpelling (word:="norepinephrine")


'Executes a method; ignores return value.
Application.CheckSpelling word:="norepinephrine"

TodaysDate = Date                 'Calls the built-in Date function
                                  'and stores the return value,
                                  'but no arguments passed.
```

Identifying Variables and Arguments Before Using Them

When you write Visual Basic code, it isn't necessary to identify variables and arguments before you use them. For example, in the following **Function** procedure, the argument is `num`, the variable is `TempVal`, and `SafeSqr` is another variable that is also the return value. Visual Basic recognizes and handles these items automatically. That's what makes creating procedures in Visual Basic so easy.

```
'Ensures that the argument is positive
'before returning the square root
Function SafeSqr(num)
    TempVal = Abs(num)
    SafeSqr = Sqr(TempVal)
End Function
```

However, there are advantages to declaring variables and arguments in advance. To *declare* a variable or other item means to make Visual Basic aware of it.

Why Identify a Variable or Argument in Advance?

Although Visual Basic takes care of most declarations and data types automatically, you can often speed up your code, conserve space, and avoid errors by declaring these items yourself.

In the preceding example, Visual Basic automatically recognizes the `TempVal` variable, so you can use it as if you had explicitly declared it. Although this is convenient, it can lead to subtle errors in your code. For example, suppose that the variable is misspelled as `TemVal` in the next-to-last line. Because `TemVal` is zero when it's created and its square root is zero, the function always returns zero rather than the correct value, but Visual Basic does not generate errors in this instance.

When Visual Basic encounters a new name, it can't determine whether you actually meant that name to be a new variable or you just mistyped an existing variable name, so it creates a new variable with the new name. To avoid the problem of misnaming variables, you can have Visual Basic generate an error message whenever it encounters a name not previously declared explicitly as a variable. To do this, place the following statement at the top of your Visual Basic module.

```
Option Explicit
```

Now you must declare variables explicitly. Within a procedure, you declare a variable explicitly with the **Dim** statement, supplying a name for the variable.

Dim *variablename*

The revised version of the SafeSqr function includes a declared variable.

```
Option Explicit

Function SafeSqr(num)
    Dim TempVal
    TempVal = Abs(num)
    SafeSqr = Sqr(TempVal)
End Function
```

If you misspell the `TempVal` variable as `TemVal`, Visual Basic displays an error message when you run the code (or when you enter the function in a cell). If you don't use **Option Explicit** at the top of your module, you can still explicitly declare some variables; the other variables are automatically declared implicitly.

Even when you choose not to use **Option Explicit** and let Visual Basic declare variables implicitly, sometimes you might still want to declare some variables explicitly with the **Dim** statement. This helps you remember what your variables are and can make your code easier to read.

Declaring Variables Explicitly All the Time to Catch Errors

Because the **Option Explicit** statement helps you catch errors, you might want to use it with all your code. By setting an environment option, you can have Visual Basic add **Option Explicit** to every module you create.

▶ **To insert Option Explicit automatically in all new modules**

1. From the Tools menu, choose Options.
2. Select the Module General tab.
3. Select the Require Variable Declaration check box.
4. Choose the OK button.

Note You must place the **Option Explicit** statement in the Declarations section (the top of the module) of every module for which you want Visual Basic to enforce explicit variable declarations. If you select the Require Variable Declaration check box, Visual Basic inserts **Option Explicit** in all new modules but does not add it to existing code. You must manually add **Option Explicit** to existing modules.

Using the Variant Data Type to Store Different Types of Information

All variables have a data type. A *data type* is the characteristic of a variable or other item that determines what kind of data it can hold (such as an integer, a string, or a date).

By default, if you don't supply a data type, a variable is given the **Variant** data type. The **Variant** data type can store many kinds of data—numbers, strings of text, or dates and times. The **Variant** data type makes it possible for you to not declare variables at all and for your variables to contain different types of values while your code is running.

Nearly all the variables you've worked with until now have been variables of the **Variant** data type (usually referred to by the shorter phrase "**Variant** variables").

Variant variables are like cells on a worksheet—the type of data they contain depends on the value you've most recently put there. First you might put a number in a cell, then a date, and then text. You can do the same thing to **Variant** variables.

For example, in the following **Function** procedure, X contains several types of data.

```
Function ReturnDataType(SomeValue)
    X = SomeValue                 'X is a Variant containing some value.
    X = "Hello World"             'X is a Variant containing a string.
    X = #12-24-1994#              'X is a Variant containing a date.
    X = 10                        'X is a Variant containing a number.
    ReturnDataType = TypeName(X) 'The name returned is Integer, but
                                  'the variable is still a Variant.
End Function
```

Visual Basic automatically makes mathematical adjustments when working with **Variant** variables.

```
Dim SomeValue        ' Variant by default.
SomeValue = "17"     ' SomeValue contains "17" (a two-character string).
' Change the numeric value of SomeValue to 2.
SomeValue = SomeValue - 15
' Change SomeValue to contain "U2" (a two-character string).
SomeValue = "U" & SomeValue
```

You can perform operations on **Variant** variables without much concern for what kind of data they actually contain. However, if you perform arithmetic operations or functions on a **Variant**, it must contain a number or something that can be interpreted as a number. For example, you cannot perform any arithmetic operations on the value `"U2"` even though it contains a numeric character, because the entire value is not a valid number. Likewise, you cannot perform any calculations on the value "1040EZ," but you can perform calculations on the values "+10" or "–.7E6" because they are valid numbers.

For more information about the **Variant** data type, see "More About Variant and Specific Data Types" later in this chapter, or search the online Visual Basic Reference for *Variant*.

Specifying the Type of Data That a Variable Can Contain

Variant variables are a very handy feature in Visual Basic because they let you concentrate on writing your code, without worrying about the type of data a variable can contain. But sometimes you want to declare variables of a specific type. For example, if you know that a variable is always an integer, you can avoid accidentally assigning it a string value. Also, declaring specific types of variables makes your code faster, more efficient, and easier to debug.

To specify that a variable must be of a specific type, declare it explicitly using the **Dim** statement, supplying a name for the variable and a data type name.

Dim *variablename* **As** *type*

The **As** *type* is optional (you can explicitly declare variables without specifying a type, which makes them the **Variant** type). You usually place all variable declarations for a procedure after the first line of the procedure (the Definition line). Following is a revised SafeSqr function that converts the `num` argument to an integer and assumes that `TempVal` should be an integer as well.

```
Function SafeSqr(num)
    'Explicitly declares TempVal to contain only an integer
    Dim TempVal as Integer
    TempVal = Abs(CInt(num))
    SafeSqr = Sqr(TempVal)
End Function
```

The preceding example uses the **Integer** data type, but there are several other specific data types. The following table shows the specific data types Visual Basic supports, including the storage size and range of each data type.

Specific data type	Storage size	Range
Boolean	2 bytes	**True** or **False**
Integer	2 bytes	–32,768 through 32,767

Specific data type	Storage size	Range
Long (long integer)	4 bytes	–2,147,483,648 through 2,147,483,647
Single (single-precision floating-point)	4 bytes	–3.402823E38 through –1.401298E–45 for negative values; 1.401298E–45 through 3.402823E38 for positive values
Double (double-precision floating-point)	8 bytes	–1.79769313486232E308 through –4.94065645841247E–324 for negative values; 4.94065645841247E–324 through 1.79769313486232E308 for positive values
Currency (scaled integer)	8 bytes	–922,337,203,685,477.5808 through 922,337,203,685,477.5807
Date	8 bytes	January 1, 0100, through December 31, 9999
String	1 byte per character	0 through 65,535 (longer on some systems)

Because Visual Basic provides specific data types, you can include specific type declarations in your code, as shown in these examples.

```
Dim Cost As Currency
Dim DidErrorOccur As Boolean
Dim ClientName As String
Dim DueDate As Date
Dim NumberOfGuests As Integer
Dim NumberOfGalaxies As Long
Dim SampleWeight As Single
Dim WeightOfTheUniverse As Double
```

In addition to the preceding specific types, Visual Basic provides other data types with special characteristics. These data types are listed in the following table.

Data type	Storage size	Range
Object	4 bytes	Any **Object** reference.
Variant	As appropriate	Any numeric value up through the range of a **Double** or any character text.
User-defined (using **Type**)	Number required by elements	The range of each element is the same as the range of its fundamental data type, listed in the previous table.

You've already seen the **Variant** data type; it and the others are described in more detail later in this chapter. In addition, the **Object** data type is demonstrated in Chapter 5, "Working with Objects in Visual Basic." You can also search the online Visual Basic Reference for information about specific data types.

Declaring Data Types of Multiple Variables on One Line

The **Dim** statement can combine multiple declarations on one line.

```
Dim I As Integer, Amt As Double
Dim YourName As String, BillsPaid As Currency
Dim Test, J As Integer, Amount
```

Note In the preceding example, the variables `Test` and `Amount` are of the **Variant** data type. Programmers with experience in other languages might have expected all variables on the same line to have the same specified type (in this case, **Integer**).

Checking the Type of Data Stored in a Variable

Checking the data type of a variable can help you avoid errors. If you want to test whether a variable is of a particular type, use the **TypeName** function, which returns the name of the data type of the variable.

```
If TypeName(NumberOfUnits) = "Integer" Then
    ' Do something to an integer
End If
```

For a list of some of the specific names that **TypeName** returns, search the online Visual Basic Reference for *TypeName*.

An error occurs if you attempt to perform a mathematical operation or function on a **Variant** variable that does not contain a number or something that can be interpreted as a number (such as a date and time value or a string containing a number). For this reason, you often want to determine if a **Variant** variable contains a value that can be used successfully as a number — the **IsNumeric** function performs this task.

```
Do
    AnyNumber = InputBox("Enter a number")
Loop Until IsNumeric(anyNumber)
MsgBox "The square root is: " & Sqr(AnyNumber)
```

IsNumeric returns **True** if its argument is any type of number (integer, long, single, or double). For more information about **IsNumeric** or similar functions, search the online Visual Basic Reference for *IsNumeric*.

Specifying the Type of Data That an Argument Can Contain

Like other variables, the arguments for procedures you write have the **Variant** data type by default. However, you can declare other data types for arguments by including **As** *type* after each argument in the first line of the procedure (the Definition line). For example, the following **Function** procedure requires that the first argument be a **String** and the second an **Integer**.

```
Function Reverse (S As String, n As Integer)
    ' Reverses the first n characters in S.
    Dim Temp As String, I As Integer
    If n > Len(S) Then n = Len(S)
    For I = n To 1 Step -1
        Temp = Temp & Mid(S, I, 1)
    Next
    Reverse = Temp & Right(S, Len(S) - n)
End Function
```

You might not yet understand all the code in this example; it reverses the order of the first *n* characters in whatever string you pass it. If you want, type the example in a Visual Basic module and enter `=Reverse("Just a test",4)` in a cell on a worksheet. The result will be "tsuJ a test."

Note Experienced programmers might distinguish between arguments (values that are passed to a procedure when it is called) and parameters (variable names that appear in the procedure declaration statement). In this manual, the term *argument* is used for both types of values.

Specifying the Type of Data That a Function Procedure Can Return

Because all Visual Basic functions return values, they also have a data type. Like variables, **Function** procedures return the **Variant** type by default. If you don't set a return value for the **Function** procedure, it returns a **Variant** containing the **Empty** value. However, a **Function** procedure can be declared to return a particular type. In the following example, the Reverse function could be declared to return a string.

```
Function Reverse (S As String, n As Integer) As String
```

In this example, if you don't assign a return value, the function returns a zero-length string (`""`). Similarly, if a function is declared to return a numeric type (such as **Integer** or **Double**), it returns zero unless you explicitly assign a return value.

As with variables, you don't have to declare a data type for a **Function** procedure because the default **Variant** data type handles most types of data automatically. However, Visual Basic can work more efficiently with functions if you explicitly declare a data type for them.

Assigning an Object to a Variable

In addition to storing values, a variable can refer to an object. You assign an object to a variable for the same reasons you assign any value to a variable — variable names are often shorter and easier to remember than the values they contain (or, in this case, the objects to which they refer), and variables can be changed to refer to other objects while your code is running.

To assign any object that is recognized by Visual Basic to a variable, use the **Set** statement. Only **Variant** variables or variables declared as **Object** or as a specific type of object can refer to objects. You will learn about specific objects later in this section.

```
Dim MyRectangle              'A Variant variable
Dim AnyRectangle As Object   'An Object (any type of object) variable

Set MyRectangle = ActiveSheet.DrawingObjects("Rectangle 1")
Set AnyRectangle = ActiveSheet.DrawingObjects("Rectangle 2")
```

Assigning Cells and Ranges of Cells (as Objects) to Variables

One of the most common objects you assign to a variable is a reference to a cell or range of cells. Cells are members of the Range object type.

When you set a variable equal to a cell reference without using the **Set** statement, the variable contains the value stored in the cell, not an actual reference.

```
'Stores the value of cell A1 in xRef
xRef = Worksheets(1).Cells(1, 1)
```

The same variable used with the **Set** statement refers to the actual cell rather than the cell's value.

```
'xRef now refers to the cell itself
Set xRef = Worksheets(1).Cells(1, 1)
```

In the preceding example, you can treat `xRef` exactly as you would treat cell A1. For example, you can set several of its properties and even store a value in the cell.

```
Set xRef = Worksheets(1).Cells(1, 1)
xRef.Font.Italic = True
xRef.Font.Bold = True
```

```
xRef.Font.Name = "Arial"
xRef.Font.Size = 14
xRef.Value = xRef.Offset(1, 0).Value
```

Or you can use the more convenient **With** statement.

```
With xRef
    .Font.Italic = True
    .Font.Bold = True
    .Font.Name = "Arial"
    .Font.Size = 14
    .Value = xRef.Offset(1, 0).Value
End With
```

The complete syntax of **Set** includes the keyword **Nothing**.

Set *objectVar* = {*objectExpression* | **Nothing**}

The braces and bar indicate that you choose one of the two options *objectExpression* or the keyword **Nothing**. Setting an object variable equal to **Nothing** discontinues the association of *objectVar* with any specific object. This frees the memory resources that Visual Basic had previously reserved for the referenced object and prevents you from accidentally changing the object by changing the variable. Using **Nothing** is usually optional because Visual Basic automatically frees variables as necessary.

```
Set xRef = Nothing
```

For more information about the keyword **Nothing**, search the online Visual Basic Reference for *Nothing*.

Assigning General and Specific Objects to Variables and Arguments

You can declare a variable so that it must refer to an object in general; to a specific type of Microsoft Excel object, such as a Range, Button, Rectangle, or Worksheet; or to any object linking and embedding (OLE) Automation object. (OLE Automation objects are described in Chapter 10, "Controlling and Communicating with Other Applications.")

It is a good idea to declare specific object types instead of the general **Object** type whenever you can. Declaring specific object types provides automatic type checking, faster code, and improved readability. Following are some examples of specific object type declarations.

```
Dim AnyObject As Object 'This variable can refer to any object
Dim CostRange As Range
Dim MyButton As Button
Dim SummarySheet As Worksheet
```

```
Set AnyObject = ActiveSheet.DrawingObjects("Rectangle 1")
Set CostRange = Worksheets("Cost Range").Range(Cells(1, 1), Cells(3, 4))
Set MyButton = ActiveSheet.DrawingObjects("Button 4")
Set SummarySheet = Worksheets("Summary")
```

For a complete list of the specific Microsoft Excel objects that a variable can refer to, search the online Visual Basic Reference for *Objects*. Notice that some objects are plural (collections), while others are singular (individual objects). For example, you can declare SummarySheets as Worksheets and SummarySheet as Worksheet.

```
Dim SummarySheets As Worksheets  'Declared as a collection of worksheets
Dim SummarySheet As Worksheet    'Declared as a Worksheet object
```

Now you can set SummarySheets to the entire collection of worksheets in a particular workbook and SummarySheet to one sheet.

```
Set SummarySheets = Workbooks("SUMMBOOK.XLS").Worksheets
Set SummarySheet = Workbooks("SUMMBOOK.XLS").Worksheets("Summary")
```

For more information about objects, see Chapter 5, "Working with Objects in Visual Basic."

In procedures, you can also require that an argument be an object. In this case, the argument works exactly the same way as a variable. The following example demonstrates the previous `xRef` example in a **Sub** procedure.

```
Sub FormatAndCopy(xRef As Range)
    With xRef
        .Font.Italic = True
        .Font.Bold = True
        .Font.Name = "Arial"
        .Font.Size = 14
        .Value = xRef.Offset(1, 0).Value
    End With
End Sub
```

Note Variables track the references to which they are assigned. For example, if you use `Set RowsRef = Rows(2)`, which makes `RowsRef` refer to row 2, and a new row is inserted at the top of the worksheet, `RowsRef` refers to row 3.

Calculating Values with Operators in an Expression

An expression is a combination of functions, numbers, variables, and operators that yields a value. The functions include all built-in Visual Basic and worksheet functions, and add-in and user-defined functions. (To make add-in functions available to Visual Basic, you must define a reference to the workbook containing

them. For more information about add-in functions, see "Creating an Add-in Application" in Chapter 13 in this book or Chapter 37, "Installing Add-in Features," in the *Microsoft Excel User's Guide.*) There are four types of operators:

- Arithmetic (+, –, /, *, ^, \, and **Mod**).
- Concatenation (**&**, also called the text operator).
- Logical (**And**, **Eqv**, **Imp**, **Not**, **Or**, and **Xor**).
- Comparison (=, >, <, >=, <=, and < >).

  Two other operators also fall into the comparison operator category: the equivalence operator (**Is**) and the similarity operator (**Like**).

The arithmetic, comparison, and concatenation operators work the same way in Visual Basic expressions as they do in worksheet formulas. Logical operators are described in the next section, "Using Logical Operators to Calculate True and False Expressions." All the arithmetic and comparison operators except \ and **Mod** are described in Chapter 10, "Creating Formulas and Links," in the *Microsoft Excel User's Guide.*

The \ operator divides two numbers and returns an integer result. Before the division, the two numbers are rounded to integers. In the following assignment statement, `Result` will contain the number 3.

```
Result = 100.4 \ 32.3
```

The **Mod** operator divides two numbers and returns only the integer remainder. In the following assignment statement, `Result` will contain the number 4.

```
Result = 100.4 Mod 32.3
```

For complete information about all the operators, search the online Visual Basic Reference for *Operators.* For more information about the order in which logical operations are tested, search the Visual Basic Reference for *Operators,* and then select the *Operator Precedence* topic.

Using Logical Operators to Calculate True and False Expressions

Logical operators usually compare two expressions, each of which evaluates to **True** or **False**. The comparison is itself an expression that evaluates to **True** or **False** based on the rules of the particular logical operator, as shown in the following table.

Syntax	Evaluates to
expression1 **And** *expression2*	**True** if both expressions are **True**; **False** otherwise.
expression1 **Eqv** *expression2*	**True** if both expressions are the same (both **True** or both **False**); **False** otherwise.

Syntax	Evaluates to
expression1 **Imp** *expression2*	**True** unless *expression1* is **True** and *expression2* is **False**.
Not *expression*	**True** if *expression* is **False**; **False** if it is **True**.
expression1 **Or** *expression2*	**True** if one or both expressions are **True**.
expression1 **Xor** *expression2*	**False** if both expressions are the same; **True** otherwise.

The following example displays a message that depends on the value of variables A, B, and C. If A = 10, B = 8, and C = 6, both expressions evaluate to **True**. Because both expressions are **True**, the **And** expression is also **True**.

```
A = 10: B = 8: C = 6          ' Assign values.
If (A > B) And (B > C) Then  ' Evaluate expressions.
    Msg = "Both expressions are True."
Else
    Msg = "One or both expressions are False."
End If
MsgBox Msg                    ' Display results.
```

In the following example, the **And** operator checks to see if the value of the Cost variable is between two values. The values are defined by constants.

```
Const MAXCOST = 1000, MINCOST = 250
Dim Cost As Currency, Revenue As Currency
Dim ProfitRef As Range

Sub CheckCosts()
Set ProfitRef = Range("Profit")
Cost = Range("Cost").Value
Revenue = Range("Revenue").Value
    If Cost < MAXCOST And Cost > MINCOST Then
        ProfitRef.Value = Revenue - Cost
    Else
        MsgBox "The cost is out of range. No profit calculated."
        ProfitRef.Value = ""
    End If
End Sub
```

The **And**, **Not**, and **Or** operators are similar to the AND, NOT, and OR built-in worksheet functions, but their syntaxes are different. **Eqv**, **Imp**, and **Xor** are available only in Visual Basic; they are not directly available to Microsoft Excel worksheets.

Tip If you use the **Eqv**, **Imp**, and **Xor** operators often, you can make them available to worksheets as user-defined functions. For example:

```
Function XorFunction (argument1, argument2)
    XorFunction = argument1 Xor argument2
End Function
```

The logical operators can work with the **Null** value as well as **True** and **False** values. For more information about logical operators, search the online Visual Basic Reference for *Logical operators*. See also "Other Types of Information Stored in Variant Variables" later in this chapter.

Using the Equivalence Operator (Is) to Compare Objects

The **Is** operator is used to test whether two variables refer to the same object.

result = *object1* **Is** *object2*

If they do refer to the same object, the expression returns **True**, and the objects are said to be logically *equivalent*. It does not matter if the object variables are declared as **Variant**, **Object**, or a specific object type.

Using the Similarity Operator (Like) to Compare Strings

The **Like** operator compares two strings.

result = *string* **Like** *pattern*

In the preceding syntax, *pattern* is a pattern of text that can be within *string* and can include wildcard characters. For example, the following statement returns **True**.

```
Test = "Mystery" Like "M*y"
```

If *string* matches *pattern*, *result* is **True**; if there is no match, *result* is **False**; and if either *string* or *pattern* is a **Null**, *result* is also a **Null**. (*String* can be any string expression.) The case sensitivity and character sort order of the **Like** operator depend on the setting of the **Option Compare** statement. Unless otherwise specified, the default string-comparison method for each module is **Option Compare Binary**; that is, string comparisons are case-sensitive. For more information, search the online Visual Basic Reference for *Option Compare*.

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features make it possible for you to use wildcard characters. The wildcard characters and what they match are shown in the following table.

Character(s) in pattern	Matches in expression
?	Any single character
*	Zero or more characters

Character(s) in pattern	Matches in expression
#	Any single digit (0 through 9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

A group of one or more characters enclosed in brackets ([ ]) can be used to match any single character in an expression and can include almost any characters in the American National Standards Institute (ANSI) character set, including digits. In fact, the special characters left bracket ([), question mark (?), number sign (#), and asterisk (*) can be used to match themselves directly only by enclosing them in brackets. The right bracket (]) cannot be used within a group to match itself, but it can be used outside a group as an individual character.

For more information about the **Like** operator, search the online Visual Basic Reference for *Like*.

Using Constants to Represent Values

Often, you'll find that your code contains constant values that appear over and over again. Or you might find that the code depends on certain numbers that are difficult to remember—numbers that, by themselves, have no obvious meaning.

In these cases, you can greatly improve the readability of your code—and make it easier to maintain—by using constants. A *constant* is a meaningful name used in place of a value. Although a constant resembles a variable, you can't modify a constant or assign a new value to it as you can to a variable. Many constants are built into Visual Basic; you can also define your own.

Working with Built-in Constants

Visual Basic provides many built-in constants that are recognized as keywords. The constants that are used with Microsoft Excel objects are preceded by the letters "xl." Constants that are used with other Visual Basic statements and functions are usually preceded by the letters "vb."

The following statements show how you can use built-in constants.

```
'Sets the calculation mode to manual
Application.Calculation = xlManual

'Retrieves the currency symbol from the International Property array
CurrencySymbol = Application.International(xlCurrencyCode)
```

```
Response = MsgBox("Do you want to continue?", vbYesNo)
'vbYesNo creates a message box with Yes and No buttons.
If Response = vbYes Then
    Msg = "You chose Yes."  'vbYes is the code for the Yes button.
Else
    Msg = "You chose No."
End If
MsgBox Msg
```

Built-in constants usually represent numbers. What the numbers are doesn't matter because Visual Basic keeps a record of this information for you. (xlManual is 2; xlCurrencyCode is 25; vbYesNo is 4; vbYes is 6.) Using constant names is preferable to using numbers because the names are easier to remember and clearer within code.

Individual topics in the online Visual Basic Reference mention relevant constants. For example, if you search the Visual Basic Reference for *Calculation property*, the topic description mentions the xlManual constant and other constants related to setting the calculation mode. The topic for **MsgBox** includes a table of constants used for creating different types of message boxes and constants that correspond to selecting different buttons.

Creating Your Own Constants

You can define your own constants using the **Const** statement.

Const *CONSTANTNAME* = *expression*

The argument *CONSTANTNAME* is any valid name (the rules are the same as the rules for creating variable names described in "More About Procedures" later in this chapter), and *expression* is composed of numeric or string constants and built-in functions and operators. It is traditional programming practice to use all uppercase letters for user-defined constants.

You can also declare your constant as a specific data type. Although this is not usually necessary, it can improve the readability of your code and the precision of some mathematical operations (for example, a constant that is a **Double** provides greater precision than one that is a **Single**).

Const *CONSTANTNAME* **As** *type* = *expression*

Because constants are usually used by more than one procedure in your module, place all the **Const** statements together at the top of your module above the first line of the procedure (the Definition line). (You can also place them within procedures to make them available only to that procedure, and you can declare them as public constants to make them available to all procedures in all modules. For more information about the availability of variables, which also applies to constants, see

"Making Your Variables Available Within Procedures, Modules, or Publicly" later in this chapter.)

```
'Top of Module
'
'Define constants for this module
'These constants represent a mathematical or date/time quantity:
Const PI = 3.14159265
Const MAX_PLANETS = 9
Const RELEASE_DATE = #1/1/94#
'
'These constants are strings:
Const VERSION = "07.10.A"
Const CODE_NAME = "Enigma"

'Begin procedure definitions
Sub FirstSub () 'First procedure in the module
    .
    .
    .
```

You can place more than one constant declaration on a single line, if you separate them with the appropriate list separators.

```
Const PI = 3.14, MAX_PLANETS = 9, AVOGADRO = 6.0225E+23
```

The expression on the right side of the equal sign (=) is often a number or literal string, but it can also be an expression with operators that results in a number or string (although that expression can't contain calls to procedures).

You can even define constants in terms of previously defined constants.

```
Const PI2 = PI * 2
```

Creating Simpler Statements with Named Arguments

For most built-in functions, statements, and methods, Visual Basic provides the option of using named arguments as a shortcut for typing many argument values and commas (or other list separators). A *named argument* is an argument name that Visual Basic recognizes. Instead of providing a value for each comma-delimited argument in the order in which they are expected by the syntax, you can assign values to specific named arguments. For example, the **ApplyNames** method has the following seven arguments.

*object.***ApplyNames(*names, ignoreRelativeAbsolute, useRowColumnNames, omitColumn, omitRow, order, appendLast*)**

Without named arguments, you might apply the name `Sales` to the range specified by `NameRef` as shown in the following example.

```
Set NameRef = Worksheets("Sheet1").Range("A1:C3")
NameRef.ApplyNames "Sales",,,,,,True
```

With named arguments, you can provide any or all of the arguments, in any order, by assigning a value to the named argument using a colon followed by an equal sign (:=) and placing that assignment in any sequence delimited by the list separator character. The following revision of the previous example uses named arguments to apply the name `Sales` to the same range.

```
Set NameRef = Worksheets("Sheet1").Range("A1:C3")
NameRef.ApplyNames appendLast:=True, names:="Sales"
```

Notice that the arguments in the second example are in reverse order.

You can use named arguments with the procedures you create, too. Visual Basic automatically associates argument names with their corresponding procedures. This is especially useful if your procedures have several optional arguments that you do not always need to specify.

Important You cannot use named arguments to avoid entering required arguments. Named arguments make it possible for you to identify specific arguments in any order, but you can omit only the optional arguments. To see which arguments are required and which are optional, search the online Visual Basic Reference for the name of the function, statement, or method you plan to use.

Learning More About Visual Basic Programming

Visual Basic is a powerful programming language with many features, but you don't have to learn about all of them before putting Visual Basic to work. The following sections explain data types, procedures, and arrays in greater detail.

- More About Variant and Specific Data Types
- Other Types of Information Stored in Variant Variables
- Converting One Data Type to Another
- Creating Your Own Data Types
- Making Your Variables Available Within Procedures, Modules, or Publicly
- How Long Do Variables Retain Their Values?
- Making Your Constants Available Within Procedures, Modules, or Publicly
- More About Procedures

- Storing Values in Visual Basic Arrays
- Using Property Procedures

If you don't need this information, you can proceed to Chapter 7, "Controlling How Your Code Runs."

More About Variant and Specific Data Types

The **Variant** data type, which was described in "Using the Variant Data Type to Store Different Types of Information" earlier in this chapter, handles many types of data and converts between them automatically. If you want to create concise, fast code, however, you might want to use specific data types. The characteristics and advantages of both the **Variant** data type and some specific data types are described in the following sections.

Numeric Types

If you know that a variable always stores whole numbers (such as 12) rather than real numbers (numbers with a fractional amount, such as 3.57), declare the variable as an **Integer** or **Long** type. Operations are faster with integers, and an integer consumes less memory than a **Variant**. Integers are especially useful as the counter variables in **For...Next** loops.

If the variable contains a fraction, declare it as a **Single**, **Double**, or **Currency** variable. The **Currency** data type supports up to 4 digits to the right of the decimal point and 15 to the left; it is a fast and accurate fixed-point data type suitable for monetary calculations. Floating-point (**Single** and **Double**) numbers have much larger ranges than **Currency** but are subject to small rounding errors.

All numeric variables can be assigned to each other and to variables of the **Variant** type. Visual Basic rounds off (rather than truncates) the fractional part of a floating-point number before assigning it to an integer.

Numeric Values Stored in Variant Variables

When you store numbers in a **Variant** variable, Visual Basic uses the most compact representation possible. So if you store a small number without a decimal fraction, the **Variant** uses an **Integer** representation for the value. If you then assign a larger number, Visual Basic will use a **Long** or (if the number is very large or has a fractional component) **Double** representation.

If you assign a **Variant** containing a number to a string variable or property, Visual Basic automatically converts the representation of the number to a string. If you want to explicitly convert a number to a string, use the **CStr** function. You can also use the **Format** function, which provides formatting such as currency, thousands separator, and decimal separator symbols. (The **CStr** and **Format** functions

automatically use the decimal separator and other symbols according to the country settings of your operating system.)

For more information about converting values, search the online Visual Basic Reference for *Conversion*. For more information about the **Format** function, search the online Visual Basic Reference for *Format*.

String Types

By default, a string (or text) variable or argument is a *variable-length string*; the string grows or shrinks as you assign new text to it. You can also declare strings that have a fixed length. You specify a *fixed-length string* with the following syntax.

String * *size*

The following example declares a string that is always 50 characters long.

```
Dim EmpName As String * 50
```

If you assign a string of fewer than 50 characters, `EmpName` is padded with enough trailing spaces to total 50 characters. If you assign a string that is too long for the fixed-length string, Visual Basic truncates the characters.

```
Dim String4 As String * 4, S As String
S = "Database"
Debug.Print S
String4 = S
Debug.Print String4
```

The preceding code produces the following output in the Immediate pane of the Debug window. (For more information about the Debug window, see "Using the Debug Window" in Chapter 8.)

```
Database
Data
```

Because fixed-length strings are padded with trailing spaces, you might find the **Trim** and **RTrim** functions useful when working with these strings. For more information about **Trim** or **RTrim**, search the online Visual Basic Reference for *Trim* or *RTrim*.

Note The kind of **Option Compare** statement you use with a Visual Basic module —**Option Compare Binary** or **Option Compare Text**—determines the results of single-letter comparisons in equality statements between strings, between strings and **Variant** variables, and between **Variant** variables containing strings.

By default, Visual Basic uses **Option Compare Binary**, which results in string comparisons based on a sort order derived from the internal binary representations of the characters. **Option Compare Text** results in string comparisons based on a case-insensitive textual sort order determined by your system's locale. For more information about binary and text comparisons, and in particular how those comparisons change in other languages, search the online Visual Basic Reference for *Option Compare.*

Strings Stored in Variant Variables

The result of the + operator varies when used with different **Variant** values. If both of the **Variants** contain numbers, then the + operator performs addition. If both of the **Variants** contain strings, then the + operator performs string concatenation. But if one of the values is represented as a number and the other is represented as a string, Visual Basic first attempts to convert the string into a number. If the conversion is successful, then the + operator adds the two values; if the conversion is unsuccessful, it concatenates the number with the string to produce a new string.

When you want to be sure that concatenation occurs regardless of the representation of the value in the variables, use the **&** operator.

You should usually use the **&** operator, instead of the + operator, for text concatenation. When typing your code, be careful to leave a space between any variable name and the **&** operator. If you do not leave a space, Visual Basic interprets **&** as the type-declaration character (for long integers) for the variable name. For more information about the type-declaration character, search the online Visual Basic Reference for *Type-declaration character.*

Dates and Times (the Date Type)

The information in this section explains how Visual Basic handles dates and times, which differs from how Microsoft Excel worksheets handle dates and times.

In Visual Basic, **Date** variables are stored as 64-bit (8-byte) numbers that represent dates ranging from January 1, 0100, through December 31, 9999, and times from 0:00:00 through 23:59:59. Any recognizable literal date values can be assigned to **Date** variables. Literal dates must be enclosed within number sign characters (#)—for example, #January 1, 1993# or #1 Jan 93#. You can compare a **Variant** containing a date or time value with a literal date.

```
If SomeDate > #3/6/93# Then
If SomeDate > #3/6/93 1:20pm# Then
```

Visual Basic accepts a wide variety of date and time formats in literals. The following are all valid date and time values.

```
SomeDate = #3-6-93 13:20#
SomeDate = #March 27, 1993 1:20am#
SomeDate = #Apr-2-93#
SomeDate = #4 April 1993#
```

Date variables display dates according to the short date format recognized by your computer. Times are displayed according to the time format (either the 12- or 24-hour format) recognized by your computer.

When other numeric data types are converted to **Date**, values to the left of the decimal represent date information while values to the right of the decimal represent time. Midnight is 0, and midday is 0.5. Negative whole numbers represent dates before December 30, 1899.

Variant variables can also contain date and time values, and the same general characteristics apply to them that apply to specific **Date** data types, but consider the following important note.

Important Always try to declare any variable or argument that contains a date, or a user-defined function that returns a date, with the **Date** data type.

```
Dim TodaysDate As Date
```

If you don't, Visual Basic won't recognize dates stored in worksheet cells unless they are formatted with a built-in date format. Also, when a user-defined function returns a date to a worksheet cell, Microsoft Excel applies a date format to the cell automatically if the variable has been declared with the **Date** data type.

Several functions return date and time values.

```
Sub DateTimeDemo ()
    Dim RightNow, DaysLeft, HoursLeft, MinutesLeft
    RightNow = Now    ' Now returns the current date/time.
    DaysLeft = Int(DateSerial(Year(RightNow) + 1, 1, 1) - RightNow)
    HoursLeft = 24 - Hour(RightNow)
    MinutesLeft = 60 - Minute(RightNow)
    MsgBox DaysLeft & " days left in the year."
    MsgBox HoursLeft & " hours left in the day."
    MsgBox MinutesLeft & " minutes left in the hour."
End Sub
```

You can perform math on date and time values as well. Adding or subtracting integers adds or subtracts days; adding or subtracting fractions adds or subtracts time as fractions of a day. Thus, adding 20 adds 20 days, while adding 1/24 adds one hour. Similarly, the value of one minute is 1/1440, and one second is 1/86400.

For more information about various date and time topics, search the online Visual Basic Reference for *Date*.

Note The decimal separator, list separator, currency symbol, and date format that you use when you're writing or editing code are determined by the Language/Country settings on the Module General tab of the Options dialog box (Tools menu). The decimal separator, list separator, currency symbol, and date formats that users see when they run your code are determined by the country settings of their operating system.

The code examples in this book assume you use U.S. country settings. For more information about writing applications for use in other countries, see Appendix A, "Writing Code for International Use."

Booleans

Boolean variables are stored as 16-bit (2-byte) numbers, but they can be only **True** or **False**. When you perform comparisons in Visual Basic, the **True** and **False** values are often implied. For example, the following two lines of code are equivalent.

```
If (Revenue > Cost) Then MsgBox "Good News!"
If (Revenue > Cost) = True Then MsgBox "Good News!"
```

When other numeric data types are converted to **Boolean** values, zero becomes **False** while all other values become **True**. When **Boolean** values are converted to other data types, **False** becomes zero while **True** becomes –1. When you compare **Boolean** values to numbers, the **Boolean** is converted first. For example, `1 = True` results in **False** because **True** is converted to –1, and `1 = -1` results in **False**.

Objects

Object variables are stored as 32-bit (4-byte) addresses that refer to objects within an application. A variable declared as an **Object** is one that can subsequently be assigned (using the **Set** statement) to refer to any actual object recognized by the application.

Other Types of Information Stored in Variant Variables

As previously stated, the **Variant** data type can store many kinds of data—numbers, strings of text, Booleans, objects, or dates and times, and so on—all of which are also specific data types. There are other types of information that can be stored in **Variant** variables, as explained in the following sections.

The Empty Value

Sometimes you need to know whether a value has ever been assigned to a variable since the variable was created or whether a cell is empty. A **Variant** variable has the **Empty** value before it is assigned a value. The **Empty** value is a special value different from zero, a zero-length string (**""**), or the **Null** value. You can test for the **Empty** value with the **IsEmpty** function.

```
If IsEmpty(Z) Then Z = 0
If IsEmpty(Cells(1, 1).Value) Then Cells(1, 1).Value = 0
```

When a **Variant** contains the **Empty** value, you can use it in expressions; it is treated as either zero or a zero-length string depending on the expression.

The **Empty** value disappears as soon as any value is assigned to a **Variant** variable (including the value of zero, the zero-length string, or the **Null** value). You can set a **Variant** variable back to the **Empty** value by assigning another **Empty Variant** variable to it or by assigning the **Empty** keyword to it.

The Null Value

The **Variant** data type can contain one other special value: **Null**. **Null** is commonly used in database applications to indicate unknown or missing data. Because of the way in which it is used in databases, **Null** has the following unique characteristics:

- Expressions involving **Null** always result in **Null**; thus, **Null** is said to *propagate* through expressions. If any part of the expression evaluates to **Null**, the entire expression evaluates to **Null**.
- Passing **Null**, a **Variant** containing **Null**, or an expression that evaluates to **Null** as an argument causes most functions to return **Null**.
- **Null** values propagate through intrinsic functions that return **Variant** data types.

You can also assign **Null** with the **Null** keyword.

```
Z = Null
```

You can use the **IsNull** function to test whether a **Variant** variable contains **Null**.

```
If IsNull(X) And IsNull(Y) Then
    Z = Null
Else
    Z = 0
End If
```

If you assign **Null** to a variable of any type other than **Variant**, a trappable error occurs. For more information about trapping errors, see Chapter 9, "Handling Errors and Error Values."

Assigning **Null** to a **Variant** variable doesn't cause an error, and **Null** does propagate through expressions involving **Variant** variables (though **Null** does not propagate through certain functions). You can return **Null** from any **Function** procedure with a **Variant** return value.

Variables are not set to **Null** unless you explicitly assign **Null** to them, so if you don't use **Null** in your application you don't have to write code that tests for and handles it.

For more information about **Null** and testing for **Null** values, search the online Visual Basic Reference for *IsNull.*

The Error Subtype

If you create your own error values using the **CVErr** function and you want to store them in variables, those variables must be **Variant** variables. Once you've assigned error values to those **Variant** variables, they are said to be of the **Error** subtype. In the following example, `MyError` is a **Variant** of subtype **Error**.

```
Dim MyError
MyError = CVErr(2010)
```

Even though `MyError` is a **Variant**, if you use it as an argument to the **TypeName** function, **TypeName** returns "Error," not "Variant." Using the **TypeName** function in this situation tells you whether a variable contains a user-defined error. You can also use the **IsError** function to test for error values.

For more information about user-defined errors, see "Creating Error Values That Don't Interrupt Your Code" in Chapter 9 or search the online Visual Basic Reference for *CVErr* or *IsError.*

Other Important Characteristics of Variants

Variants will be discussed throughout the remainder of this chapter and this manual as necessary. For now, note the following additional characteristics of **Variants**.

- **Variant** variables can refer to objects.
- A single **Variant** variable can contain an entire array. The array can contain other **Variant** variables, or variables of other data types. For more information about arrays, see "Storing Values in Visual Basic Arrays" later in this chapter.
- **Variant** variables cannot contain user-defined data types (records) or arrays that contain user-defined data types. For more information about user-defined data types, see "Creating Your Own Data Types" later in this chapter.

Converting One Data Type to Another

Sometimes you want to use a specific representation for a number. You can convert a variable to a particular type using the **C***type* functions. For example, you might want a **Variant** variable to store a numeric value as **Currency** to avoid round-off errors in later calculations. Visual Basic provides several conversion functions that you can use to convert values into a specific type. To convert a value to **Currency**, for example, you use the **CCur** function.

```
PayPerWeek = CCur(hours * hourlyPay)
```

You can use the **CDate** function to convert a text value into a date and time value. For example, the following code tests the **Value** property of a cell with **IsDate**. If the property contains text that can be considered a valid date or time, Visual Basic converts the text into a date and computes the days left until the end of the year.

```
Dim SomeDate, DaysLeft
Dim CellRef, CellRef2 As Range
Set CellRef = Worksheets(1).Cells(1, 1)
Set CellRef2 = Worksheets(2).Cells(2, 1)

If IsDate(CellRef.Value) Then
    SomeDate = CDate(CellRef.Value)
    Daysleft = DateSerial(Year(SomeDate) + 1, 1, 1) - SomeDate
    CellRef2.Value = DaysLeft & " days left in the year."
Else
    MsgBox CellRef.Value & " is not a valid date."
End If
```

Visual Basic includes specific conversion functions for **Boolean**, **Currency**, **Double**, **Single**, **Long**, **Integer**, **String**, **Date**, and **Variant** types. For more information about conversion functions, search the online Visual Basic Reference for *Conversion*. For more information about dates, search the Visual Basic Reference for *Date*, *IsDate*, and *DateSerial*.

Creating Your Own Data Types

You can combine variables of several different types to create your own data types. User-defined types are useful when you want to create a single variable that records several related pieces of information.

Note Experienced programmers might be more familiar with the term *structures*, also known as *structs* in the C programming language and *records* in Pascal. *User-defined type* is used in this manual to mean the same thing.

You create a user-defined type with the **Type** statement, which must be placed at the top of a Visual Basic module. User-defined types are always public. (Public variables are described in the following section, "Making Your Variables Available Within Procedures, Modules, or Publicly.") However, a variable that you declare as a user-defined type can be local, module-level, or public in all modules. For example, you can create the following user-defined type that records information about a computer system.

```
' Declarations
Type SystemInfo
    CPU As Variant
    Memory As Long
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

You can declare local, module-level, or public variables of the same user-defined type.

```
Dim MySystem As SystemInfo, YourSystem As SystemInfo
```

Assigning and returning values from the elements of this variable is similar to setting and getting properties.

```
MySystem.CPU = "486"
If MySystem.PurchaseDate > #1/1/94# Then
```

You can also assign one variable to another if they are both of the same user-defined type. This assigns all the elements of one variable to the same elements in the other variable.

```
YourSystem = MySystem
```

A user-defined type can contain an ordinary (fixed-size) array.

```
Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives(25) As String     'Fixed-size array
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

It can also contain a dynamic array. (For more information about dynamic arrays, search the online Visual Basic Reference for *ReDim.*)

```
Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives() As String       'Dynamic array
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

You can access the values in an array within a user-defined type in the same way that you access the property of an object.

```
Dim MySystem As SystemInfo
MySystem.DiskDrives(1) = "1.44 MB"
```

You can also declare an array of user-defined types.

```
Dim AllSystems(100) As SystemInfo
```

Follow the same rules to access the components of the data structure.

```
AllSystems(5).CPU = "486D50"
AllSystems(X).DiskDrives(2) = "100M SCSI"
```

Nesting of data structures can get as complex as you like. (Nesting means to place one structure inside another.) In fact, user-defined types can contain other user-defined types.

```
Type DriveInfo
    Type As String
    Size As Long
End Type
```

```
Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives(26) As DriveInfo
    Cost As Currency
    PurchaseDate As Variant
End Type

Dim AllSystems(100) As SystemInfo
AllSystems(1).DiskDrives(0).Type = "Floppy"
```

You can declare procedure arguments with a user-defined type.

```
Sub FillSystem (SomeSystem As SystemInfo)
    SomeSystem.CPU = lstCPU.Text
    SomeSystem.Memory = txtMemory.Text
    SomeSystem.Cost = txtCost.Text
    SomeSystem.PurchaseDate = Now
End Sub
```

You can return user-defined types from functions, and you can pass a user-defined type variable to a procedure as one of the arguments. User-defined types are always passed by reference, so the procedure can modify the argument and return it to the procedure that called it, as illustrated in the previous example. For more information about passing by reference, see "More About Procedures" later in this chapter.

Note The total size of any record in a user-defined data type cannot exceed 65,535 bytes.

Making Your Variables Available Within Procedures, Modules, or Publicly

When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. Sometimes, however, you need to use a variable with a broader scope, such as one whose value is available to all the procedures within the same Visual Basic module, or even to all the procedures in your workbook. The *scope* of a variable is its availability throughout your modules. Using Visual Basic, you can specify the scope of a variable when you declare it.

Depending on how it is declared, a variable is scoped one of three ways.

Scope	Declaration
Local	**Dim** or **Static** within the procedure
Module	**Dim**, **Static**, or **Private** at the top of the module
Public	**Public** at the top of the module

Variables Used Within a Procedure (Local Variables)

A *local variable* is recognized only within the procedure in which it appears. It's a good choice for any kind of temporary calculation. A dozen procedures can all have a variable called `Temp`, for instance, but as long as each `Temp` is local, each procedure recognizes only its own variable. A procedure can alter its local `Temp` variable without affecting `Temp` variables in other procedures.

```
Sub Procedure1()
    Dim Temp As Integer ' Local variable, not the same as Temp below
.
.
.
End Sub

Sub Procedure2()
    Dim Temp As Integer ' Local variable, not the same as Temp above
.
.
.
End Sub
```

A procedure can even call itself (a technique called *recursion*), and each invocation of the procedure gets its own copy of the local variables in that procedure. Local variables declared with **Dim** remain in existence only as long as the procedure is executing. Local variables declared with **Static** (described in "How Long Do Variables Retain Their Values?" later in this chapter) remain in existence the entire time Visual Basic is running. They are reset whenever an untrapped run-time error occurs, you halt Visual Basic or quit Microsoft Excel, or you change the module containing the variable. For more information about error handling, see Chapter 9, "Handling Errors and Error Values."

Note Implicitly declared variables (allowed if the **Option Explicit** statement is not in effect) always have local scope. For more information about **Option Explicit**, see "Declaring Variables Explicitly All the Time to Catch Errors" earlier in this chapter.

Variables Used Within a Module (Module-Level Variables)

A *module-level variable* shares information among all the procedures in a module. It's available to all the procedures in that module, but not to code in other modules. You create module-level variables by declaring them with the **Dim** or **Private** statement at the top of the module, above the first procedure definition. (At the module level, there is no difference between **Private** and **Dim**, but you might prefer to use **Private** because it readily contrasts **Private** and **Public** variables and makes your code easier to read.) Module-level variables remain in existence while Visual Basic is running until the module in which they are defined is edited.

```
Dim NumberOfEmployees As Integer     'Module-level variable
Private WeightOfUniverse As Double   'Module-level variable

Sub Procedure1()
    Dim Temp As Integer 'Procedure-level (local) variable
.
.
.
```

Variables Used by All Modules (Public Variables)

A *public variable* has the broadest scope of all. The values in **Public** variables are available to every procedure in every module in all workbooks (provided the workbooks are selected in the Available References box of the References dialog box, Tools menu). For more information about adding references to workbooks, see "Using OLE Automation with Applications" in Chapter 10.

Note Experienced users might be more familiar with the term *global* to describe universally available variables, constants, and procedures. In the present context, global and public have similar meanings; both refer to these universally available elements.

Like module-level variables, **Public** variables are declared in the Declarations section of a module (the top section, above procedure definitions); you cannot declare module-level or **Public** variables in a procedure. However, you declare **Public** variables with the **Public** statement instead of with the **Dim** or **Private** statement that module-level variables use.

```
Public NumberOfEmployees As Integer  'Public variable
Public WeightOfUniverse As Double    'Public variable

Sub Procedure1()
    Dim Temp As Integer 'Procedure-level (local) variable
.
.
.
```

You can declare your **Public** variables in any module. For the purposes of organizing your code, however, it's often a good idea to precede the variable name with its module qualifier. For example, if you're in a module named Costs and you want to access the `ClothingSales` variable in the Revenues module, you can use the following code.

```
Total = AllSales + Revenues.ClothingSales
```

This helps you avoid accessing **Public** variables with the same name that are declared in two different modules, something that can cause an error when you run your code.

A **Public** variable exists and retains its value from the time a value is assigned to the variable to until execution stops.

Note To create variables or procedures that are available to all modules in the workbook but not to other workbooks, use the **Option Private Module** statement in the module and declare the variable using **Public** (procedures are **Public** by default). If you don't include **Option Private Module**, then other workbooks can access the **Public** variables and procedures in the module, provided that a reference has been established between the two workbooks. For more information about adding references to workbooks, see "Using OLE Automation with Applications" in Chapter 10.

Using Two Variables with the Same Name

A variable cannot change scope while your code is running. However, you can have a variable with the same name at a different scope. For example, you might have a public variable called `Temp` and then, within a procedure, declare a local variable called `Temp`. References to the name `Temp` within the procedure access the local variable; references to `Temp` outside the procedure access the public variable.

In general, when variables have the same name but different scope, the more local variable is always accessed before less local variables.

Having variables with the same name can be confusing and lead to subtle bugs in your code, so it's a good idea to use unique names, or use module qualifiers if you're using variables with the same names in different modules.

Using Variables and Procedures with the Same Name

The names of your module-level and public variables can also conflict with the names of your procedures. A variable in the module cannot have the same name as any procedures or types defined in the module. It can, however, have the same name as **Public** procedures, types, or variables defined in other modules. In this case, when the variable is accessed from another module, it must be qualified with the module name (for example, `CostMod.TotalCosts`).

When the variable is accessed from the module in which it is defined, however, you do not have to include the module name. **Private** procedures, variables, and user-defined types have module-level scope for the module in which they are defined and, therefore, cannot have the same name as another procedure, module variable, or user-defined type in the same module.

How Long Do Variables Retain Their Values?

In addition to scope, variables also have a lifetime. The values in module-level and public variables are preserved while Visual Basic is running. However, local variables declared with **Dim** exist only while the procedure in which they are declared is running. Usually, when a procedure is finished running, the values of its local variables are not preserved and the memory used by the local variables is cleared. The next time the procedure is run, all its local variables are reinitialized.

However, you can preserve the value of a local variable by making the variable static. Use the **Static** keyword to declare one or more variables inside a procedure, exactly as you would with the **Dim** statement.

```
Static Depth
```

For example, the following function calculates a running total by adding a new value to the total of previous values stored in the **Static** variable `Accumulate`.

```
Function RunningTotal(num)
    Static Accumulate
    Accumulate = Accumulate + num
    RunningTotal = Accumulate
End Function
```

If `Accumulate` is declared with **Dim** instead of **Static**, the previous accumulated values are not preserved across calls to the function, and the function simply returns the same value with which it was called.

You can produce the same result by declaring `Accumulate` in the Declarations section at the top of the module, making it a module-level variable. Once you change the scope of a variable this way, however, the procedure no longer has exclusive access to it. Because other procedures can access the value of the variable and change it, the running totals might be unreliable and the code is more difficult to maintain.

Preserving the Values of All Local Variables in a Procedure

To make all local variables in a procedure static, place the **Static** keyword at the beginning of a procedure heading.

```
Static Function RunningTotal(num)
```

This makes all the local variables in the procedure static, regardless of whether they are declared with **Static** or **Dim** or declared implicitly. You can place **Static** in front of any procedure heading, including those that are also declared as **Private**.

Making Your Constants Available Within Procedures, Modules, or Publicly

A **Const** statement has scope just as a variable declaration does, and the following rules apply to both:

- To create a constant that exists only within a procedure, declare it within that procedure.
- To create a constant available to all procedures within a module but not to any code outside that module, declare the constant in the Declarations section of the module or declare it with the **Private** keyword.
- To create a constant available throughout all workbooks, declare the constant in the Declarations section of a module and place the keyword **Public** before **Const**.

There is one possible problem to consider when you're using public constants. Because constants can be defined in terms of other constants, you must be careful not to set up a circular reference between two or more constants. A *circular reference* occurs when you have two or more public constants, each of which is defined in terms of the other.

```
' In Module 1:
Public Const A = B * 2

' In Module 2:
Public Const B = A / 2
```

When a circular reference occurs, Visual Basic generates an error when you attempt to run your application. You cannot run your code until you resolve the circular reference.

More About Procedures

This section explains more about how you name procedures, how you pass information to procedures with arguments, and how Visual Basic returns values from **Function** procedures.

Rules for Naming Procedures, Arguments, Variables, and Constants

When you name your procedures, arguments, variables, and constants, keep in mind the following rules:

- Visual Basic ignores uppercase and lowercase but preserves whatever case you use the last time you typed a particular argument, variable, constant, or procedure name.
- The first character must be a letter.
- In the names of **Function** procedures, you cannot use a name that resembles a cell reference in A1- or R1C1-style notation, or any restricted Visual Basic keyword. For more information about restricted and unrestricted keywords, search the online Visual Basic Reference for *Restricted keywords*.

Changing a Variable's Value When It's Passed to a Procedure (by Value vs. by Reference)

When you call a procedure and you use variables as arguments to the procedure, you can have the procedure change the value of those variables.

A variable's value can be changed by a procedure when it's passed to that procedure by reference. Passing *by reference* is a way of passing an argument to a procedure that allows the procedure to have access to the actual variable in its memory address location. As a result, the variable's value can be permanently changed by the procedure to which it is passed. Passing by reference is the default in Visual Basic.

By contrast, when the variable is passed *by value,* only a copy of the variable is passed to the procedure. If the procedure changes that value, the change affects only the copy and not the original variable itself. This is important in the Reverse procedure described in "Specifying the Type of Data That an Argument Can Contain" earlier in this chapter; if the second argument is not declared by **ByVal**, bugs can appear in the code.

```
Function Reverse (S As String, ByVal n As Integer)
    ' Reverses the first n characters in S.
    Dim Temp As String, I As Integer
    If n > Len(S) Then n = Len(S)    'Changing the value of n can change
                                     'a value in the function call.
    For I = n To 1 Step -1
        Temp = Temp & Mid(S, I, 1)
    Next
    Reverse = Temp & Right(S, Len(S) - n)
End Function
```

The preceding function can change the value of `I` in the following function call because `I` (the second argument in the call) corresponds to `n` (the second argument in the **Function** procedure).

```
R = Reverse(S, I)
```

Now suppose the second argument in Reverse wasn't declared **ByVal** and you called it.

```
Dim I As Integer, S As String
I = 10
S = "Testing"
R = Reverse(S, I)
' Now I = 7 (the length of S).
```

The function modifies its arguments—something you don't usually expect to happen. To avoid this kind of side effect in any procedures that modify their arguments, declare those arguments with **ByVal**.

Passing arguments by reference is a way to make **Sub** procedures act like **Function** procedures except that you change variables rather than return values. For example, the following two lines of code can both change the value of `MyVariable` if the value is passed to the MySub **Sub** procedure by reference.

```
MyVariable = MyFunction()    'MyFunction returns some value
MySub MyVariable             'MySub changes variables passed by reference
```

For more information about passing by value and by reference, search the online Visual Basic Reference for *Passing arguments*, *ByVal*, and *ByRef*.

Passing Microsoft Excel Arrays to a Procedure

When you pass a Microsoft Excel array as an argument to a Visual Basic procedure, it is converted automatically to a Visual Basic array. For example, on a Microsoft Excel worksheet, you can call a user-defined function named ProcessFood that has a **Variant** argument `Food`.

```
Function ProcessFood(Food)
```

`Food` becomes a Visual Basic array (of **Variant** variables) when you pass a Microsoft Excel array to the function, as shown in the following formula that you can enter in a cell on a worksheet.

```
=ProcessFood({"Apples",7,19})
```

Note The decimal separator, list separator, currency symbol, and date format that you use when you're writing or editing code are determined by the Language/Country settings on the Module General tab of the Options dialog box (Tools menu). The decimal separator, list separator, currency symbol, and date formats that users see when they run your code are determined by the country settings of their operating system.

The code examples in this book assume you use U.S. country settings. For more information about writing applications for use in other countries, see Appendix A, "Writing Code for International Use."

The conversion of a Microsoft Excel array to a Visual Basic array follows these rules:

- A horizontal Microsoft Excel array is converted into a one-dimensional Visual Basic array.

  For example, `=ProcessFood({"Apples",7,19})` creates an array in the user-defined function with Food(1) = "Apples", Food(2) = 7, and Food(3) = 19.

- A vertical Microsoft Excel array is converted into a one-dimensional Visual Basic array.

  For example, `=ProcessFood({"Apples";7;19})` creates an array in the user-defined function with Food(1) = "Apples", Food(2) = 7, and Food(3) = 19.

- A two-dimensional Microsoft Excel array is converted to a two-dimensional Visual Basic array, and both array indexes (row and column) are used.

  For example, `=ProcessFood({"Apples",7;19,"Peaches"})` creates an array in the user-defined function with Food(1,1) = "Apples", Food(1,2) = 7, Food(2,1) = 19, and Food(2,2) = "Peaches".

When returning arrays to Microsoft Excel, a one-dimensional Visual Basic array is converted to a horizontal Microsoft Excel array. A two-dimensional Visual Basic array is converted to a two-dimensional Microsoft Excel array.

For more information about Visual Basic arrays, see "Storing Values in Visual Basic Arrays" later in this chapter. For more information about Microsoft Excel arrays, see "Working with Arrays" in Chapter 10 in the *Microsoft Excel User's Guide*.

Creating a Procedure with Optional Arguments

You can specify that arguments to a procedure are optional by placing the **Optional** keyword in the argument list. If you specify an optional argument, all subsequent arguments in the argument list must also be optional and declared with the **Optional** keyword. The optional arguments must be of the **Variant** data type.

By using the following revision of the Reverse function, the user can omit the last argument. Notice that you test for the missing argument using the **IsMissing** function.

```
Function Reverse (S As String, Optional ByVal n)
    ' Reverses the first n characters in S.
    ' If n is omitted, all characters are reversed
    Dim Temp As String, I As Integer
    If IsMissing(n) Then n = Len(S)
    If  n > Len(S) Then n = Len(S)
    For I = n To 1 Step -1
        Temp = Temp & Mid(S, I, 1)
    Next
    Reverse = Temp & Right(S, Len(S) - n)
End Function
```

Creating a Procedure with an Indeterminate Number of Arguments

You can create **Function** procedures that accept any number of arguments of the **Variant** data type. SUM, AVERAGE, and MEDIAN are examples of built-in functions that accept a variable number of arguments. To specify that an argument can include an indeterminate list of values, declare it using the **ParamArray** keyword. The argument is an array. To retrieve individual values, use an index or the **For Each...Next** statement as you do with any array.

Note The lower bound of the **ParamArray** array (zero-based or one-based) is determined by the array base of the Visual Basic module. Visual Basic defaults to a zero-based array unless you specify otherwise by typing **Option Base 1** in the Declarations section of the module. For more information about **Option Base**, search the online Visual Basic Reference for *Option Base.*

For example, suppose you wanted to use the MULTINOMIAL worksheet function without installing the entire Analysis ToolPak; you can create your own version of the function. (The Analysis ToolPak is an add-in macro that provides financial, statistical, and engineering functions. For more information, see Chapter 31, "Statistical Analysis of Data," in the *Microsoft Excel User's Guide.*)

```
'Returns ratio of the factorial sum to the factorial product
'(num1 + num2 + ...)! / (num1! * num2! * ...)
Function MyMultinomial (ParamArray Numbers())
    For Each x in Numbers
        s = s + x
    Next x
    Numerator = Application.Fact(s)
    Denominator = 1
    For Each x in Numbers
        Denominator = Denominator * Application.Fact(x)
    Next x
    MyMultinomial = Numerator / Denominator
End Function
```

More About Returning Values from a User-Defined Function

When you return ranges, arrays, or text from user-defined functions, keep the following restrictions in mind:

- You can return an array only if you declare the data type of the return value as **Variant**.
- Text returned from a user-defined function to a Microsoft Excel object can be no longer than 255 characters.

Creating User-Defined Functions That Recalculate Automatically

Use the **Volatile** method to make your user-defined functions volatile. A *volatile* function is recalculated every time a calculation occurs anywhere on the worksheet. (Usually, a worksheet recalculates a cell containing a nonvolatile user-defined function only when any part of the complete formula in the cell is recalculated.) Call the **Volatile** method as the first statement in the procedure. For more information about the **Volatile** method, search the online Visual Basic Reference for *Volatile method.*

Storing Values in Visual Basic Arrays

Visual Basic arrays are similar to Microsoft Excel arrays and array formulas. For more information about Microsoft Excel arrays, see "What Is an Array Constant?" and "Working with Arrays" in Chapter 10 in the *Microsoft Excel User's Guide.*

In Visual Basic, you can use arrays to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create shorter and simpler code in many situations, because you can set up loops that deal efficiently with your variables. Arrays have both upper and lower bounds, and the elements of the array are contiguous within those bounds. Because Visual Basic allocates space for each index number, avoid declaring an array larger than you need it to be. You can change the size or dimensions of an array with **ReDim**. For more information about the **ReDim** statement, see "Changing the Size of an Array While Your Code is Running" later in this chapter.

All the elements in an array have the same data type. Of course, when the data type is **Variant**, the individual elements can contain different kinds of data (strings, numbers, or date and time values). You can declare an array of any of the primary data types, including user-defined types (described in "Creating Your Own Data Types" earlier in this chapter) and object variables.

The three ways to declare a fixed-size array, depending on the scope you want the array to have, are as follows:

- To create a *local array*, declare the array within a procedure using the **Dim** statement.
- To create a *module-level array*, use the **Dim** or **Private** statement in the Declarations section of a Visual Basic module to declare the array.
- To create a *public array,* use the **Public** statement in the Declarations section of a module to declare the array.

Local, module-level, and public scope are described in "Making Your Variables Available Within Procedures, Modules, or Publicly" earlier in this chapter. Scope applies to arrays in the same way that it applies to variables.

There are additional rules when you create a *dynamic array* (an array whose size can change while your code is running). These rules are discussed in "Preserving the Values in Dynamic Arrays" later in this section.

When declaring an array, follow the array name by its upper bound enclosed in parentheses. For example, the following array declarations can appear in the Declarations section of a module.

```
Dim Counters(14) As Integer
Dim Sums(20) As Double
```

To create a public array, you simply use **Public** in place of **Dim**.

```
Public Counters(14) As Integer
Public Sums(20) As Double
```

The same declarations within a procedure can also use **Static**.

```
Static Counters(14) As Integer
Static Sums(20) As Double
```

The first declaration creates an array with 15 elements, with index numbers running from 0 through 14. The second creates an array with 21 elements, with index numbers running from 0 through 20. The default lower bound is 0 (for a zero-based array). However, you can change the default lower bound to 1 (for a one-based array) by placing an **Option Base** statement in the Declarations section of a module.

```
Option Base 1
```

Tip You might want to include `Option Base 1` in your modules all the time. Microsoft Excel collections are always one-based, and any arrays that Microsoft Excel methods or properties return are also one-based. If you assign a one-based array to a zero-based array, the same information is in both arrays, but each element is off by one index. This can make your code confusing and hard to debug.

Another way to specify a lower bound is to provide it explicitly (as a long integer) using the **To** keyword.

```
Dim Counters(1 To 15) As Integer
Dim Sums(100 To 120) As String
```

In the preceding declarations, the index numbers of `Counters` run from 1 through 15, and the index numbers of `Sums` run from 100 through 120.

Loops often provide an efficient way to manipulate arrays. For example, the following **For...Next** loop initializes all elements in the array to 5.

```
Static Counters(1 To 15) As Integer
For X = 1 To 15
    Counters(X) = 5
Next
```

Storing Arrays in Variants

You can declare an array of **Variant** variables, or with the **Array** function you can create a **Variant** variable that contains an array of any type of variables.

```
Dim MyArray(10) 'Creates an array of ten Variant variables.

Dim WeekDay
WeekDay = Array ("Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun")
'Creates a Variant variable containing an array
```

Working with Multidimensional Arrays

With Visual Basic, you can declare arrays of up to 60 dimensions. For example, the following statement declares a two-dimensional 10-by-10 array within a procedure (the array is zero-based).

```
Static MatrixA(9, 9) As Double
```

Either or both dimensions can be declared with explicit lower bounds.

```
Static MatrixA(1 To 10, 1 To 10) As Double
```

You can efficiently process a multidimensional array by using nested **For** loops. For example, the following statements initialize each element in `MatrixA` to a value based on its location in the array.

```
Dim I As Integer, J As Integer
Static MatrixA(1 To 10, 1 To 10) As Double
For I = 1 To 10
    For J = 1 To 10
        MatrixA(I, J) = I * 10 + J
    Next J
Next I
```

You can also extend an array to more than two dimensions.

```
Dim MultiD(3, 1 To 10, 1 To 15)
```

The preceding declaration creates an array that has three dimensions with sizes 4 by 10 by 15. The total number of elements is the product of these three dimensions, or 600.

Note When you start adding dimensions to an array, the total storage needed by the array increases dramatically, so use multidimensional arrays with care. Be especially careful with **Variant** arrays, because **Variant** arrays are larger than other data types.

Changing the Size of an Array While Your Code Is Running

Sometimes you might not know exactly how large to make an array, and you want to be able to change its size while your code is running. Dynamic arrays can be resized at any time. Dynamic arrays help you manage memory efficiently. For example, you can use a large array for a short time and then free memory to the system when you're no longer using the array.

▶ To create a dynamic array

1. Declare the array with a **Static** or **Dim** statement in a procedure (if you want the array to be local), a **Dim** statement at the module level (if you want the array to be module-level), or a **Public** statement (if you want the array to be public).
2. Declare the array as dynamic by giving it an empty dimension list.

   ```
   Dim DynArray()
   ```

3. Allocate the actual number of elements with a **ReDim** statement. You can do this only in a procedure, not at the module level.

   In the following code, `X` is a variable whose value is determined while your code is running. It can be used to change the previously defined maximum number of elements.

   ```
   ReDim DynArray(X + 1)
   ```

The **ReDim** statement can appear only in a procedure. Unlike the **Dim** and **Static** statements, **ReDim** is an executable statement—it makes the application carry out an action when you run your code.

The **ReDim** statement supports the same syntax described for fixed arrays. Each **ReDim** can change the number of elements in addition to the lower and upper bounds for each dimension.

For example, the dynamic array `Matrix1` is created by first declaring it at the module level.

```
Dim Matrix1() As Integer
```

A procedure then allocates space for the array with an empty dimension list.

```
Sub CalcValuesNow ()
    .
    .
    .
    ReDim Matrix1(19, 29)
End Sub
```

The **ReDim** statement allocates a matrix of 20 by 30 integers (at a total size of 600 elements). Alternatively, the bounds of a dynamic array can be set using variables.

```
ReDim Matrix1(X, Y)
```

When you create a dynamic array that is local to a procedure, declaring the array with a **Dim** or **Static** statement is recommended but not required.

Preserving the Values in Dynamic Arrays

Each time you execute the **ReDim** statement, all the values currently stored in the array are lost. Visual Basic resets the values to the **Empty** value (for **Variant** arrays), to zero (for numeric arrays), to a zero-length string (for string arrays), or to **Nothing** (for arrays of objects). For more information about object arrays, see Chapter 5, "Working with Objects in Visual Basic".

This is useful when you want to prepare the array for new data, or when you want to shrink the array to take up minimal memory. However, sometimes you might want to change the size of the array without losing the data in the array. You can do this by using **ReDim** with the **Preserve** keyword. For example, you can enlarge an array by 10 elements without losing the values of the existing elements with code.

```
ReDim Preserve MyArray(UBound(MyArray) + 10)
```

Only the upper bound of the last dimension in a multidimensional array can be changed when you use the **Preserve** keyword; if you change any of the other dimensions, or the lower bound of the last dimension, a run-time error occurs. Thus, you can do this:

```
ReDim Preserve Matrix(10, UBound(Matrix, 2) + 1)
```

But you cannot do this:

```
ReDim Preserve Matrix(UBound(Matrix, 1) + 1, 10)
```

For more information about dynamic arrays, search the online Visual Basic Reference for *ReDim*. For more information about run-time errors, see Chapter 9, "Handling Errors and Error Values."

Using Property Procedures

In most cases, to set the value of a property you use the syntax described in Chapter 5, "Working with Objects in Visual Basic." However, there are some cases when you can benefit from writing procedures that you use in your code as if they were properties. For example, you might run a procedure to perform data validation or initialize a worksheet simply by assigning a value to a property you define.

In Visual Basic, you can create special procedures that work similarly to setting and returning property values. These procedures are called *property procedures*. When you create a property procedure, the new property is a property of the Visual Basic module containing the property procedure. Typically, you don't qualify the property with the name of the module containing it.

Visual Basic provides three kinds of property procedures as described in the following table.

Property procedure	Description
Property Let	A procedure that sets the value of a property
Property Get	A procedure that returns the value of a property
Property Set	A procedure that sets a reference to an object

You can create **Property Let, Property Get**, and **Property Set** procedures that share the same name. By doing this, you can create a group of procedures that work together. For example, you might write one property procedure that sets the value of a property with **Property Let,** and a second property procedure that returns the same value with **Property Get**.

Although property procedures can share a name, they do not have to perform similar or related tasks.

Creating a Procedure That Sets a Property Value

With the **Property Let** statement, you can create a procedure that sets the value of a property. If you want to validate data before you actually assign that data to a property value, for example, you can create a property procedure to do that.

For example, you might have a macro that calculates the discount price of an item, provided the amount of the discount is no more than 35 percent. Instead of assuming the discount amount is correct, you can use the **Property Let** statement to create a property procedure that checks the discount before calculating the price.

```
Property Let MarkDown(ByVal Discount)
    If Discount <= 35 Then
        ActiveCell.Formula = Price * (1 - (Discount / 100))
    Else
        MsgBox prompt:= "Discount too high.", title:= "Error"
    End If
End Property
```

The name of this property procedure is MarkDown. The code inside the procedure checks the amount of the discount and then calculates the discount price for valid discount amounts or displays an error message for invalid discounts.

Notice that the procedure takes an argument. Property procedures created with the **Property Let** or **Property Set** statement must define at least one argument to receive the value of the property setting. (The **Property Set** statement is discussed later in this chapter.) Procedures call **Property Let** procedures by setting the value of the name of the property procedure, as shown in the following example.

```
Prices.MarkDown = 25
```

In this example, the MarkDown property procedure is a property of the Prices module. Including the name of the Visual Basic module in the syntax is optional. The value of the property becomes the value of the `Discount` argument in the property procedure.

Property Let procedures can take more than one argument. When calling a property procedure created with **Property Let**, the last argument always appears on the right side of the equal sign. The other arguments appear in parentheses following the name of the property. For example, assume we change the MarkDown procedure so it takes two arguments—the original price of the item and the amount of the discount. The first line of the property procedure then looks like this:

```
Property Let MarkDown(ByVal Price, ByVal Discount)
```

Calling the procedure now, you pass the Price argument inside the parentheses (using a variable in this case).

```
Prices.MarkDown(Retail) = 25
```

For more information about the **Property Let** statement, search the online Visual Basic Reference for *Property Let.*

Creating a Procedure That Returns a Property Value

With the **Property Get** statement, you can create a property procedure that returns a property value. **Property Get** procedures are very similar to **Function** procedures in that both return a value. They differ only in that you can define a **Property Let** or **Property Set** procedure with the same name. Otherwise, procedures created with **Property Get** are identical to **Function** procedures.

In this example, the **Property Get** procedure returns the value of a cell.

```
Property Get CellContents()
    CellContents = Format(ActiveSheet.Range("Totals").Value, "Currency")
End Property
```

You can call this procedure with code similar to the following statement.

```
' Returning a property value with the CellContents procedure
X = CellContents
```

Notice that like a **Function** procedure, the code inside the **Property Get** procedure must set the value you want to return to the name of the procedure. Property procedures created with the **Property Get** statement can take arguments, but they don't have to.

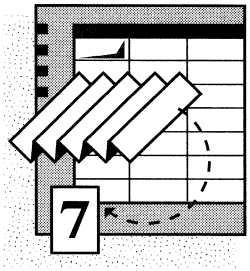
For more information about the **Property Get** statement, search the online Visual Basic Reference for *Property Get.*

Creating a Procedure That Sets an Object Reference

With the **Property Set** statement, you can create a property procedure that sets a reference to an object. You create and use property procedures using **Property Set** much as you do with the **Property Let** statement. For more information about assigning a reference to an object, see "Assigning Cells and Ranges of Cells (as Objects) to Variables" earlier in this chapter.

CHAPTER 7

Controlling How Your Code Runs



In this chapter, you will learn about *decision making*—how to write procedures that test conditions and then run only certain statements based on the results. You will also learn about *looping*—how to run a group of statements several times—and *nesting*—placing one control structure inside another.

The statements that control decision making and looping in Visual Basic are called *control structures*. Without control structures, a procedure's logic flows through statements from left to right, top to bottom. Although some very simple procedures can be written with only this unidirectional flow, most of the power and utility of any programming language comes from its ability to use control structures to change the order in which statements are run.

Contents

Running Portions of Code Based on Conditions (Decision Making)

Visual Basic procedures can test conditions and then, depending on the results of that test, perform different operations. The decision structures that Visual Basic supports include:

- **If...Then**
- **If...Then...Else**
- **Select Case**

If...Then

Use the **If...Then** syntax to run one or more statements depending on conditions encountered. You can use either a single-line syntax or a multiple-line "block" syntax.

```
If AnyDate < Now Then AnyDate = Now
```

–Or–

```
If AnyDate < Now Then
    AnyDate = Now
    MsgBox "Today's date is " & AnyDate
End If
```

Notice that the single-line form of **If...Then** does not use an **End If** statement. If you want to execute more than one line of code when the condition is **True**, you must use the multiple-line block **If...Then...End If** syntax.

If...Then...Else

Use the **If...Then...Else** syntax to define a block of statements, one of which is always run.

```
If Age < 16 Then
    MsgBox "You are not old enough for a license."
Else
    MsgBox "You can be tested for a license."
End If
```

You can add an **ElseIf** statement to test several conditions; this replaces several nested **If...Then** statements, making your code shorter and easier to read. For example, suppose you have a group of job classifications and different bonus rates are based on those job classifications. The following **Function** procedure uses a series of **ElseIf** statements to test the job classification.

```
Function Bonus(JobClass, Salary, Rating)
    If JobClass = 1 Then
        Bonus = Salary * 0.1 * Rating / 10
    ElseIf JobClass = 2 Then
        Bonus = Salary * 0.09 * Rating / 10
    ElseIf JobClass = 3 Then
        Bonus = Salary * 0.07 * Rating / 10
    Else
        Bonus = 0
    End If
End Function
```

Following is an informal version of the syntax for **If...Then...Else**. For more information about **If...Then...Else**, search the online Visual Basic Reference for *If...Then...Else.*

If *condition1* **Then**
    statements
ElseIf *condition2* **Then**
    statements
.
.
.
Else
    statements
End If

Visual Basic first tests *condition1*. If it's **False**, Visual Basic tests *condition2,* and so on, until it finds a **True** condition. When it finds a **True** condition, Visual Basic executes the corresponding statements under the condition and then goes on to execute the code following **End If**. As an option, you can include the **Else** statement, which Visual Basic executes if none of the conditions is **True**.

Note When the condition you are evaluating contains an **Or** operator, such as `If (IsError(A) Or (A>9))`, both expressions are tested, even if the first one is **True**. This behavior can affect the outcome of the statement in rare circumstances. For example, it can cause a run-time error if a variable in the second expression contains an error value.

Notice that you can have any number of **ElseIf** statements or none at all. You can include an **Else** statement whether or not you have **ElseIf** statements. You can always add more **ElseIf** statements to your **If...Then** structure. However, this can get tedious to write when each **ElseIf** statement compares the same expression to a different value. For this situation, you can use a **Select Case** decision structure.

Select Case

Visual Basic provides the **Select Case** statement as an alternative to **If...Then...ElseIf** for comparing the same expression to several different values. A **Select Case** statement provides capability similar to the **If...Then...Else** statement, but it makes code more efficient and readable.

For example, to add several more job classifications to the previous example, you can add more **ElseIf** statements, or you can write the function with **Select Case**. The following example adds several job classifications, but they represent the same **Case**.

```
Function Bonus(JobClass, Salary, Rating)
    Select Case JobClass
        Case 1
            Bonus = Salary * 0.1 * Rating / 10
        Case 2
            Bonus = Salary * 0.09 * Rating / 10
        Case 3
            Bonus = Salary * 0.07 * Rating / 10
        Case 4, 5   'The expression list can contain several values...
            Bonus = Salary * 0.05 * Rating / 5
        Case 6 To 8 '...or be a range of values
            Bonus = 150
        Case Is > 8 '...or be compared to other values
            Bonus = 100
        Case Else
            Bonus = 0
    End Select
End Function
```

Notice that the **Select Case** structure evaluates an expression once at the top of the structure. In contrast, the **If...Then...ElseIf** structure can evaluate a different expression for each **ElseIf** statement. You can replace an **If...Then...ElseIf** structure with a **Select Case** structure only if each **ElseIf** statement evaluates the same expression.

Following is an informal version of the syntax for **Select Case**. For more information, particularly about the power and flexibility of *expressionlist*, search the online Visual Basic Reference for *Select*.

Select Case *testexpression*
  Case *expressionlist1*
    statements
  Case *expressionlist2*
    statements
.
.
.
  Case Else
    statements
End Select

A **Select Case** structure works with a single *testexpression* that is evaluated once, at the top of the structure. Visual Basic then compares the result of this expression with the values for each **Case** in the structure. If there is a match, it executes the block of statements associated with that **Case**.

Each *expressionlist* is a list of one or more values. If there is more than one value in a single list, the values are separated by commas (or the appropriate list separator). If more than one **Case** matches the *testexpression*, only the statement block associated with the first matching **Case** is executed. Visual Basic executes statements in the **Case Else** statement (which is optional) if none of the *expressionlist* values matches the *testexpression*.

Running the Same Code Several Times (Looping)

You can use loop structures to execute one or more lines of code repetitively. The most important loop structures that Visual Basic supports are:

- **Do...Loop**
- **For...Next**
- **For Each...Next**

You can also perform multiple actions on the same object with the **With** statement. For more information about the **With** statement, see "Performing Multiple Actions on an Object" in Chapter 5.

Do...Loop

Use a **Do...Loop** to run a block of statements an indefinite number of times. There are several variations of the **Do...Loop** statement, but each evaluates a numeric condition to determine whether to continue running. As with **If...Then**, *condition* must be a value or expression that evaluates to **False** (zero) or to **True** (nonzero). The different variations are described in this section. For more information about the **Do...Loop** statement, search the online Visual Basic Reference for *Do...Loop*.

Checking a Condition Before Running a Loop

The following **Function** procedure counts the occurrences of a target string within another string by looping as long as the target string is found. The loop can be run any number of times, but the statements inside the loop are never run if *condition* is initially **False**.

```
Function CountStrings (longstring, target)
    position = 1
    Do While InStr(position, longstring, target) 'Returns True/False
        position = InStr(position, longstring, target) + 1
        Count = Count + 1
    Loop
    CountStrings = Count
End Function
```

If the target string doesn't occur in the other string, then **InStr** returns zero and the loop isn't executed.

Following is an informal version of the syntax for a **Do...Loop** in which the *statements* are executed as long as *condition* is **True**.

Do While *condition*
    statements
Loop

The *statements* must eventually cause *condition* to become **False**, or the loop runs forever (an *infinite loop*), a situation you should try to avoid. To stop an infinite loop, press the ESC key.

Running a Loop at Least Once Before Checking a Condition

Another variation of the **Do...Loop** statement runs the statements first and then tests *condition* after each execution. This variation guarantees that your procedure carries out each statement at least one time.

```
Do
    ProcessUserData      'Call procedure to process data at least once.
    Response = MsgBox ("Do you want to process more data?", vbYesNo)
Loop While Response = vbYes
```

Following is an informal version of the syntax for this variation of the **Do...Loop** statement.

Do
    statements
Loop While *condition*

Running a Loop While a Condition Is False

Two other variations are analogous to the previous two, except that they loop as long as *condition* is **False** rather than **True**.

```
Response = MsgBox("Do you want to process more data?", vbYesNo)
Do Until Response = vbNo
    ProcessUserData 'Call procedure to process data
    Response = MsgBox("Do you want to process more data?", vbYesNo)
Loop
```

The following example shows the version that executes each statement at least once.

```
Response = MsgBox("Do you want to process more data?", vbYesNo)
Do
    ProcessUserData 'Call procedure to process data
    Response = MsgBox("Do you want to process more data?", vbYesNo)
Loop Until Response = vbNo
```

Following are informal versions of the syntax for these variations of the **Do...Loop** statement.

Do Until *condition*
    statements
Loop

–And–

Do
    statements
Loop Until *condition*

Notice that **Do Until** *condition* is exactly equivalent to **Do While Not** *condition.*

For...Next

When you don't know how many times you need to run the statements in a loop, use **Do** loops. When you know you must execute the statements a specific number of times, use a **For** loop. Unlike a **Do** loop, a **For** loop uses a *counter* variable that increases or decreases in value during each repetition of the loop.

For example, this **Sub** procedure sounds a tone the number of times you specify.

```
Sub BeepSeveral()
    NumBeeps = InputBox("How many beeps?")
    For Counter = 1 To NumBeeps
        Beep
    Next Counter
End Sub
```

The preceding example is a simple loop. You can make the loops more flexible by having the counter step by an increment you specify. The following **Sub** procedure replaces every other value in an array with zero.

```
Sub ClearArray(ByRef ArrayToClear())
    For I = LBound(ArrayToClear) To UBound(ArrayToClear) Step 2
        ArrayToClear(i) = 0
    Next I
End Sub
```

Following is an informal version of the syntax for **For...Next**.

For *counter* = *start* **To** *end* [**Step** *increment*]
    statements
Next [*counter*]

The arguments *counter*, *start*, *end*, and *increment* are all numeric. The square brackets indicate optional elements and aren't actually typed in your code.

Note The argument *increment* can be either positive or negative. If *increment* is positive, *start* must be less than or equal to *end* or the statements in the loop won't be run. If *increment* is negative, *start* must be greater than or equal to *end* for the body of the loop to be run. If **Step** isn't set, then *increment* defaults to 1.

When Visual Basic executes a **For...Next** loop, it follows these steps:

1. Sets *counter* equal to *start.*
2. Tests to see if *counter* is greater than *end.* If so, Visual Basic exits the loop.
   (If *increment* is negative, Visual Basic tests to see if *counter* is less than *end.*)
3. Runs the *statements.*
4. Increments *counter* by 1, or by *increment* if it's specified.
5. Repeats steps 2 through 4.

For Each...Next

A **For Each...Next** loop is similar to a **For...Next** loop, but it repeats a group of statements for each element in a collection of objects or in an array, instead of repeating the statements a specified number of times. This is especially helpful if you don't know how many elements are in a collection. For more information about collections and more examples of **For Each...Next**, see Chapter 5, "Working with Objects in Visual Basic."

For example, the following **Sub** procedure closes every open workbook.

```
Sub CloseEveryWorkbook()
'Book is a variable that refers to elements in the Workbooks collection.
    For Each Book in Workbooks()
        Book.Close
    Next Book
End Sub
```

The following procedure increments the value of each cell in the selected range, if that value is a number. Remember, a range of cells is an object, and you must use the **Set** statement when you create any variable that refers to an object.

```
Sub IncrementCurrentSelection()
    Set RangeToIncrement = Selection
    'C refers to a particular cell during each loop.
    For Each C in RangeToIncrement
        If IsNumeric(C.Value) Then
            C.Value = C.Value + 1
        End If
    Next C
End Sub
```

For more information about the **Set** statement, see "Assigning an Object to a Variable" in Chapter 6.

Following is an informal version of the syntax for **For Each...Next**.

For Each *element* **In** *group*
    statements
Next *element*

When Visual Basic executes a **For Each...Next** loop, it follows these steps:

1. Defines *element* as naming the first element in *group* (provided there is at least one element).
2. Runs the *statements.*
3. Tests to see if *element* is the last element in *group.* If so, Visual Basic exits the loop.
4. Defines *element* as naming the next element in *group.*
5. Repeats steps 2 through 4.

Keep the following restrictions in mind when using **For Each...Next**:

- For collections, *element* can be only a **Variant** variable, a generic **Object** variable, or any specific OLE Automation object variable. (For information about OLE Automation objects, see Chapter 10, "Controlling and Communicating with Other Applications.") For arrays, *element* can only be a **Variant** variable.
- You cannot use the **For Each...Next** statement with an array of user-defined types because a **Variant** cannot contain a user-defined type.

For more information about **For Each...Next**, search the online Visual Basic Reference for *For.*

Placing One Control Structure Inside Another (Nesting)

As the previous example demonstrates, you can place control structures inside other control structures (for instance, an **If...Then** block within a **For Each...Next** loop within another **If...Then** block, and so on). A control structure placed inside another control structure is said to be nested.

The following **Function** procedure is similar to the IncrementCurrentSelection **Sub** procedure in the previous section, but this procedure searches a range of cells and counts the number of cells matching a value you specify.

```
Function CountValues(RangeToSearch, SearchValue)
    If TypeName(RangeToSearch) <> "Range" Then
        MsgBox "You can search only a range of cells."
    Else
        For Each C in RangeToSearch
            If C.Value = SearchValue Then
                Counter = Counter + 1
            End If
        Next C
    End If
    CountValues = Counter
End Function
```

Notice that the first **End If** closes the inner **If** loop and that the last **End If** closes the outer **If** loop. Likewise, in nested **For** and **For Each** statements, the **Next** statements automatically apply to the nearest prior **For** or **For Each** statement. Nested **Do...Loop** structures work in a similar fashion, with the innermost **Loop** statement matching the innermost **Do** statement.

Control structures in Visual Basic can be nested to as many levels as you want. It's common practice to make nested decision structures and loop structures more readable by indenting the body of the decision structure or loop, as shown in the preceding code.

Exiting Control Structures and Procedures

Usually, when you use a control structure, you provide an initial condition; the code continues in the control structure until that condition is met, and then the code exits the control structure and continues through the rest of the procedure. However, you can sometimes speed up your code by exiting a control structure early—as soon as some other condition is met.

For example, if you're searching for a value in an array using a **For...Next** loop and you find the value the first time through the loop, there's no reason to search the rest of the array. You can exit the loop immediately. Or an error or another event might occur that makes it unnecessary for a procedure to attempt some action; you can exit that procedure immediately, too. You exit a control structure or procedure early by using one of the **Exit** statements.

Exiting a Control Structure

You can use the **Exit For** statement to exit directly from a **For** or **For...Next** loop and the **Exit Do** statement to exit directly from a **Do** loop.

The following revision of the previous example counts the number of cells that have to be searched before finding the value you specify (that is, the procedure finds the value's position in the list of cells to be searched). It stops counting and searching as soon as it finds the matching value.

```
Function Position(RangeToSearch, SearchValue)
    Found = False
    For Each C in RangeToSearch
        Counter = Counter + 1
        If C.Value = SearchValue Then
            Found = True
            Exit For 'Stop counting and testing
        End If
    Next C
    If Found Then
        Position = Counter
    Else
        Position = 0
    End If
End Function
```

As the preceding example illustrates, an **Exit** statement almost always appears inside an **If** statement or **Select Case** statement nested inside the loop. This is because you should exit only under extraordinary conditions; otherwise, let the loop complete itself as usual.

Syntactically, the **Exit** statement is simple: **Exit For** can appear as many times as needed inside a **For...Next** loop or **For Each...Next** loop, and **Exit Do** can appear as many times as needed inside a **Do...Loop**.

For *counter* = *start* **To** *end* **Step** *increment*
    statements
    Exit For
    statements
Next *counter*

–Or–

For Each *element* **In** *group*
    statements
    Exit For
    statements
Next *element*

The **Exit Do** statement works with all versions of the **Do...Loop** syntax.

Do While *condition*
    statements
    Exit Do
    statements
Loop

Exiting a Procedure

The way you exit a procedure is similar to the way you exit a control structure—with **Exit Sub** and **Exit Function** statements. The syntax of these statements is similar to that of **Exit For** and **Exit Do**, described in the previous section. **Exit Sub** can appear as many times as needed, anywhere within the body of a **Sub** procedure. The same is true for **Exit Function**.

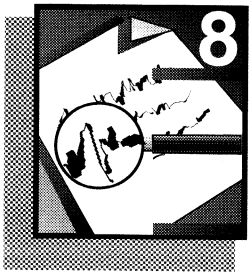
Exiting is useful when the procedure has finished everything it needs to do and can return immediately to the procedure that called it. For example, if you want to change the CountValues example so that it exits immediately if it finds the string "Bad Data," you can add an **Exit Function** statement.

```
Function CountValues(RangeToSearch, SearchValue)
    For Each C in RangeToSearch
        If C.Value = SearchValue Then
            Counter = Counter + 1
        ElseIf C.Value = "Bad Data" Then
            CountValues = Null
            Exit Function   'Stop testing and exit immediately.
        End If
    Next C
    CountValues = Counter
End Function
```

This completes your introduction to control structures in Visual Basic. With the information you've learned in this and previous chapters, you'll be able to write powerful Visual Basic code to automate your tasks. But as your code becomes more sophisticated, errors are more likely to occur. The next chapter explains how to test and debug your code when it doesn't perform as you expect.

CHAPTER 8

Testing and Debugging Your Code



Visual Basic provides tools to help analyze how your code operates. These debugging tools are particularly useful in locating the source of errors, or *bugs*. But you can also use them to experiment with changes to your code or to learn how code written by other people works.

This chapter shows you how to use the debugging tools included in Visual Basic.

Contents

How Debugging Tools Help

An incorrect result might be produced at the end of a long series of calculations. In debugging, the task is to determine where something went wrong. Perhaps you forgot to assign the correct value to a variable, chose the wrong operator, or used an incorrect formula.

There are no magic tricks to debugging; however, there are some approaches you can use that are helpful. Begin by identifying the kind of error that appears when you run your code. Then use the debugging tools described in this chapter to step through the code to identify where an error occurs. Debugging tools give you a snapshot of the current state of your code. The better you understand how your code works, the faster you can find any bugs.

These debugging tools cannot diagnose or fix errors for you. They can help you analyze how execution flows from one part of the code to another, and how variables and property settings change as statements are executed. Debugging tools are a way of looking inside your code; they can help you determine what happens and why.

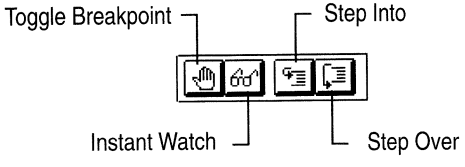
Kinds of Errors

To understand how debugging is useful, consider the kinds of errors you might encounter.

- *Language errors* result from incorrectly constructed code. You might have incorrectly typed a keyword, omitted some necessary punctuation, or used a **Next** statement without a corresponding **For** statement. Visual Basic detects these errors when you leave the line or immediately before it runs the code.
- *Run-time errors* occur, and are detected by Visual Basic, when a statement attempts an operation that is impossible to carry out. A good example of this is a reference to an object that cannot exist in the current context. For example, you might write some code that refers to a range of cells on the active sheet. If the active sheet is not a worksheet, a run-time error occurs when the code runs. The error occurs because only a worksheet can contain a range of cells.
- *Logic errors* occur when code doesn't perform the way you intended. Your code might be syntactically valid and run without performing any invalid operations and yet produce incorrect results. Only by testing the code and analyzing results can you verify that the code performs as you intended. Logic errors can also produce run-time errors.

Debugging Buttons on the Visual Basic Toolbar

Microsoft Excel provides several buttons on the Visual Basic toolbar that are very helpful when debugging. The Visual Basic toolbar appears automatically when you switch to a Visual Basic module. The following illustration shows the portion of this toolbar that includes these buttons.



The following table briefly describes the function of each button. The following sections in this chapter note instances when each of these buttons can help you debug or analyze code more efficiently.

Debugging button	Function
Toggle Breakpoint	Creates or removes a breakpoint. A breakpoint is a location in the code where Visual Basic halts execution.
Instant Watch	Displays the current value of an expression while the code is in break mode.
Step Into	Runs only the next executable line of code. If the code calls another procedure, your view into the code shifts to the called procedure until it ends.
Step Over	Runs the next executable line of code, running entire procedures when called without shifting your view of the code to the called procedure.

Using Break Mode

Break mode halts the operation of code and gives you a picture of its condition at that moment. When you enter break mode:

- The Debug window appears.
- You can use the Visual Basic toolbar, but you can't switch to other sheets in the workbook.

In break mode, variable and property settings are preserved, so you can analyze their current state. You can also make changes that affect how the code runs. When code is in break mode, you can:

- Determine which active procedures have been called.
- Watch the values of variables, properties, and statements.
- Change the values of variables and properties.
- Run Visual Basic statements immediately.
- Manually control the line-by-line execution of the code.

Entering Break Mode at a Problem Statement

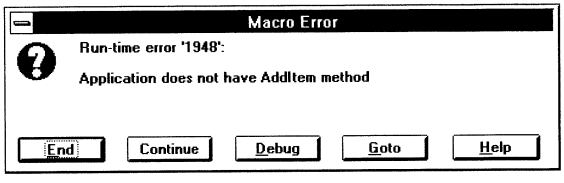
When debugging, you might want the code to halt at the place where you think the problem started. This is one reason Visual Basic provides breakpoints and **Stop** statements. A *breakpoint* defines a statement or set of conditions at which Visual Basic automatically stops execution and puts the code in break mode without running the statement containing the breakpoint. The **Stop** statement, a Visual Basic keyword, also puts Microsoft Excel in break mode.

Entering Break Mode Automatically

Visual Basic enters break mode when any of the following occurs:

- Execution reaches a line with a breakpoint.
- Execution reaches a **Stop** statement.
- A break expression defined in the Add Watch dialog box changes or becomes true, depending on how you defined it. For more information about break expressions, see "Monitoring Data with Watch Expressions" later in this chapter.
- A statement in a line of code generates an untrapped run-time error. For more information about trapping errors, see Chapter 9, "Handling Errors and Error Values."

  The following illustration shows an example of the message that appears when an error occurs. Depending on the type of error that occurs, you can quit running the code or try to debug it.



Entering Break Mode Manually

To enter break mode while the code is not running, choose the Debug Window command from the View menu. Visual Basic can also switch into break mode while the code is running. You can do this using the keyboard.

System	Press
Windows	CTRL+BREAK or ESC
Macintosh	COMMAND+PERIOD or ESC

Using a Breakpoint to Selectively Halt Execution

At run time, a breakpoint tells Visual Basic to halt just before a specific line of code. When Visual Basic executes a procedure and it encounters a line of code with a breakpoint, it switches to break mode.

Note You can set or remove a breakpoint whether you are in break mode or not. However, you cannot set a breakpoint inside an add-in application because you cannot edit add-ins. For more information about add-in applications, see Chapter 13, "Creating Automatic Procedures and Add-in Applications."

▶ **To set or remove a breakpoint**

1. With a Visual Basic module active, move the insertion point to the line of code where you want to set or remove a breakpoint.
2. From the Run menu, choose Toggle Breakpoint.

   You can also click the Toggle Breakpoint button on the Visual Basic toolbar.

When you set a breakpoint, Visual Basic highlights the selected line, using the colors you specify, and makes the text bold. For example, the following illustration shows a procedure halted by a breakpoint.

```
Sub DoubleNumber()
    Dim Number
    Number = ActiveCell.Value
    Number = Number * 2
    ActiveCell.Value = Number
End Sub
```

Identifying the Current Statement

In break mode, a rectangular outline surrounds the *current statement,* or next statement to be executed, if the module containing that statement is visible.

When the current statement also contains a breakpoint, only the rectangular outline indicates the line of code. Once the current statement moves to another line, the line with the breakpoint appears in bold and in color once again.

You can specify the color of the text and background for the current statement or a line with a breakpoint by choosing the Options command from the Tools menu, selecting the Module Format tab, and then choosing the colors you want in the Code Colors box.

Examining the Code at a Breakpoint

Once you reach a breakpoint and the code is halted, you can examine the code's current state.

A breakpoint halts the code just before executing the line that contains the breakpoint. If you want to observe what happens when the line with the breakpoint executes, you must execute at least one more statement. You can step through individual statements to accomplish this task. For more information about stepping through statements, see "Running Selected Portions of Your Code" later in this chapter.

When you are trying to isolate a problem, remember that a statement might be indirectly at fault because it assigns an incorrect value to a variable. You can examine the value of variables and properties while in break mode, using watch

expressions or the Immediate pane of the Debug window. "Monitoring Data with Watch Expressions" later in this chapter explains how to define and use watch expressions. "Testing Data and Procedures with the Immediate Pane" explains how to test expressions and examine the values of variables.

Using Stop Statements to Enter Break Mode

Placing a **Stop** statement in a procedure is an alternative to setting a breakpoint. Whenever Visual Basic encounters a **Stop** statement, it halts execution and switches to break mode. (To resume execution, choose the Continue command from the Run menu or click the Resume Macro button on the Visual Basic toolbar.) Although **Stop** statements act the same way breakpoints do, they aren't set or cleared the same way.

Note There are important differences between **Stop** statements and breakpoints. If you close and reload the current workbook, all breakpoints are cleared. But **Stop** statements stay in the code until you remove them.

Be careful not to confuse the **Stop** statement with the **End** statement. They are not interchangeable. A **Stop** statement does nothing more than temporarily halt execution by entering break mode, whereas an **End** statement halts execution completely.

Using the Debug Window

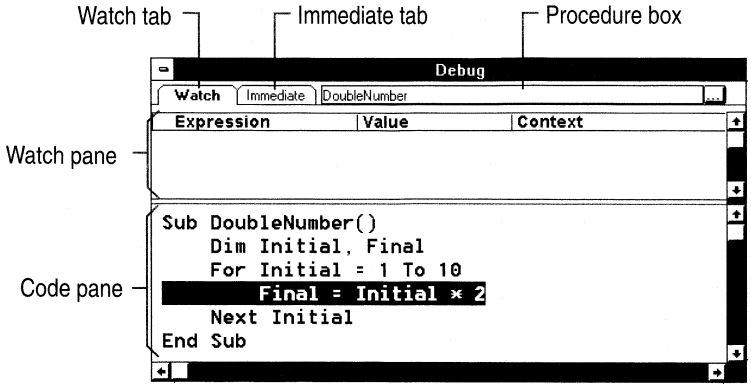
Sometimes you can find the cause of a problem by executing portions of code. But more often you have to analyze what's happening to the data as well. For example, you might isolate a problem in a variable or property with an incorrect value. Then you must determine how and why that variable or property was assigned an incorrect value.

In the Debug window, you can monitor the values of expressions and variables while stepping through the statements in your code. You can also use the Debug window to change the value of variables and properties in break mode to see how different values affect your code.

You display the Debug window by:

- Entering break mode. The Debug window automatically appears when Visual Basic enters break mode.
- Choosing the Debug Window command from the View menu.

The Debug window has two halves. Either the Watch pane or the Immediate pane appears in the upper half. The lower half displays the Code pane. The *Code pane* shows the current execution point in the code, when visible. You can scroll through the Code pane to look at other parts of the code.



The Debug window with the Watch pane on top

The *Watch pane* displays the current watch expressions—expressions whose values you decide to monitor as the code runs. The *Immediate pane* displays information that results from debugging statements in your code or which you request by typing commands directly in the panel. The first time you display the Debug window, the Immediate pane is on top; thereafter, the last pane you chose is on top.

In the Watch pane, the Context column indicates the procedure, module, or modules in which each watch expression is evaluated. The Watch pane can display a value for a watch expression only if the current statement is in the specified context. Otherwise, the Value column displays a message indicating the statement is not in context.

Running Selected Portions of Your Code

If you can identify the statement that caused an error, a single breakpoint might help you locate the problem. More often, however, you know only the general area of the code that caused the error. A breakpoint helps you isolate that problem area. You can then use Step Into and Step Over to observe the effect of each statement.

Using Step Into

Stepping is the process of executing one statement at a time. After stepping through a statement, you can observe its effect in the Debug window. When stepping into a procedure, Visual Basic executes the current statement and then advances to the next statement. After that, it switches back to break mode.

▶ **To step through code one statement at a time**

- From the Run menu, choose Step Into.

  You can also click the Step Into button on the Visual Basic toolbar.

If the statement executed is a call to another procedure in an unprotected module, you can use Step Into to step through each statement in the called procedure. When you reach the end of the called procedure, Step Into returns you to the procedure that contained the call. When the procedure being called is in an Add-In or in a module with its Visible property set to xlVeryHidden, Step Into behaves as Step Over does.

Note You can use Step Into between individual statements, even if they are on the same line. A line of code can contain two or more statements, separated by a colon (:). Visual Basic uses a rectangular outline to indicate which of the statements executes next. Breakpoints apply only to the first statement on a line.

Using Step Over

Step Over is identical to Step Into, except when the current statement contains a call to a procedure. Unlike Step Into, which steps into the called procedure, Step Over executes it as a unit and then steps to the next statement in the current procedure. Suppose, for example, that a module contains these two procedures, the first of which calls the second.

```
Sub ComputePrice()
    NumShares = InputBox("Number of shares?")
    SharePrice = InputBox("Price per share?")
    Cost = Commission(NumShares, SharePrice)
    MsgBox "The commission cost is: " & Cost
End Sub

Function Commission(SharesSold, PricePerShare)
    TotalSalePrice = SharesSold * PricePerShare
    If TotalSalePrice <= 15000 Then
        Commission = 25 + (0.03 * SharesSold)
    Else
        Commission = 25 + (0.03 * (0.9 * SharesSold))
    End If
End Function
```

If you begin stepping through the ComputePrice procedure, Step Into and Step Over behave identically until you get ready to execute this line:

```
Cost = Commission(NumShares, SharePrice)
```

If you use Step Into on this line, the Code pane of the Debug window displays the Commission procedure and sets the current statement to the beginning of that procedure. This is the better choice only if you want to analyze the code within the Commission procedure.

If you use Step Over on the same line, however, Visual Basic executes the Commission procedure without stepping into it. The Code pane displays the next line in the ComputePrice procedure, unless Commission contains a breakpoint or a **Stop** statement. Use Step Over if you want to stay at the same level of code and don't need to analyze the Commission procedure.

Tip If you inadvertently step into a lengthy procedure call, it can take a long time to get back to the procedure you left. You can get back out quickly by placing a breakpoint at the end of the procedure you stepped into, and then clicking the Resume Macro button on the Visual Basic toolbar. Once you reach the breakpoint, continue stepping to return to the original procedure.

You can alternate freely between Step Into and Step Over. The best choice depends on which portions of code you want to analyze at any given time.

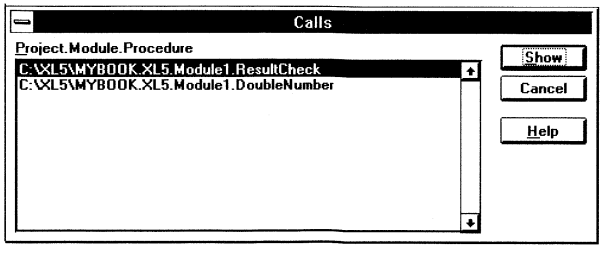
▶ **To use Step Over**

- From the Run menu, choose Step Over.

  You can also click the Step Over button on the Visual Basic toolbar.

Using the Calls Dialog Box

The Calls dialog box helps you trace the operation of code as it executes a series of procedures, especially nested procedures. For example, one procedure can call a second procedure, which can call a third procedure—all before the first procedure is completed. Such nested procedure calls can be difficult to follow.



You can display the Calls dialog box only when the code is in break mode.

▶ **To display the Calls dialog box**

- In the Debug window, choose the button to the right of the Procedure box (the box containing the name of the current procedure).

Tracing Nested Procedures

When you have a series of nested calls, the Calls dialog box displays the most recent active procedure call at the top of the list. The information given for each procedure begins with the filename of the workbook followed by the name of the module and the called procedure.

You can also use the Calls dialog box to display the statement in a procedure that passes control of the code to the next procedure in the list.

▶ **To display the statement that calls another procedure**

1. In the Calls dialog box, select the procedure call you want to display.
2. Choose the Show button.

   The procedure appears in the Code pane.

Monitoring Data with Watch Expressions

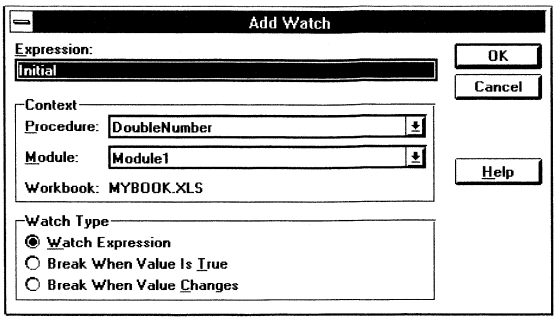
As you debug your code, you might find that a problem occurs only when a certain variable or property assumes a particular value or range of values. Or you might learn that a calculation isn't producing the desired result. Many debugging problems aren't immediately traceable to a single statement, so you might need to observe the behavior of a variable or expression throughout a procedure.

You can monitor the value of a particular variable or expression in a *watch expression*. Visual Basic monitors watch expressions for you. In break mode, watch expressions appear in the Watch pane of the Debug window, where you can observe their values.

You can also direct watch expressions to enter break mode whenever the watched expression's value changes or equals a specified value. For example, instead of using Step Into to move through hundreds of loops, you can use a watch expression to put the code in break mode when a loop counter reaches a specific value. Or you can enter break mode each time a certain variable changes value.

Adding a Watch Expression

You can add a watch expression before running a procedure or after entering break mode. You use the Add Watch dialog box to add watch expressions.



The following table describes the Add Watch dialog box controls.

Control	Description
Expression box	Where you enter the expression that the watch expression evaluates. The expression can be a variable, a property, a function call, or any other valid expression.
Context option group	Sets the scope of variables watched in the expression. Use the options in this group if you have variables of the same name with different scope. You can also restrict the scope used to watch variables defined for a specific procedure, a specific module, or globally in the code. Visual Basic can evaluate a variable in a narrow context more quickly.
Watch Type option group	Sets how Visual Basic responds to the watch expression. Visual Basic can watch the expression and display its value in the Watch pane when the code enters break mode. Or you can have the code enter break mode automatically when the expression evaluates to a true (nonzero) statement or when the value of the expression changes.

▶ **To add a watch expression**

1. From the Tools menu, choose Add Watch.
2. In the Expression box, type the expression you want to evaluate.
3. To set the scope of the expression watched, select the procedure or module name in the appropriate box.
4. To determine how you want Visual Basic to respond to the watch expression, select an option button under Watch Type.
5. Choose the OK button.

Editing or Deleting a Watch Expression

You can edit or delete any watch expression you select in the Watch pane of the Debug window. When you choose the Edit Watch command from the Tools menu, the Edit Watch dialog box appears. This dialog box looks just like the Add Watch dialog box with a Delete button added.

▶ **To edit a watch expression**

1. In the Watch pane of the Debug window, select the watch expression you want to edit.
2. From the Tools menu, choose Edit Watch.
3. In the Edit Watch dialog box, make any changes to the expression, the scope for evaluating variables, or the watch type.
4. Choose the OK button.

Tip You can also edit any watch expression displayed in the Debug window by double-clicking the watch expression in the Watch pane. Visual Basic displays the selected watch expression in the Edit Watch dialog box.

▶ **To delete a watch expression**

1. In the Watch pane of the Debug window, select the watch expression you want to delete.
2. From the Tools menu, choose Edit Watch.
3. In the Edit Watch dialog box, choose the Delete button.

Tip If your keyboard has a DEL key, you can delete a watch expression by selecting it in the Watch pane, and then pressing DEL.

Identifying Watch Types

To the left of each watch expression in the Watch pane of the Debug window is an icon that identifies the watch type of that expression. The following illustration shows the icon for each of the three watch types.

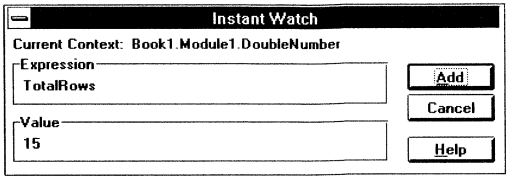
– Watch expression

– Break when expression is True

– Break when expression has changed

Using Instant Watch

While in break mode, you can check the value of an expression for which you have not defined a watch expression. You can check such expressions with the Instant Watch dialog box.



The Instant Watch dialog box displays the value of an expression that you select from:

- A Visual Basic module.
- The Code pane of the Debug window.

To continue watching this expression, choose the Add button. If Visual Basic cannot evaluate the current expression, the Add button is not available.

▶ **To add a watch expression from Instant Watch dialog box**

1. Select the expression you want to watch.
2. From the Tools menu, choose Instant Watch.

   You can also click the Instant Watch button on the Visual Basic toolbar.
3. Choose the Add button.

Testing Data and Procedures with the Immediate Pane

Sometimes when you are debugging or experimenting with code, you might want to test procedures, evaluate expressions, or assign new values to variables or properties. You can use the Immediate pane of the Debug window to accomplish these tasks. You evaluate expressions by printing their values in the Immediate pane.

Printing Information in the Immediate Pane

There are two ways to print to the Immediate pane:

- Include **Print** methods in your Visual Basic code.
- Enter **Print** methods directly in the Immediate pane.

These **Print** method techniques offer several advantages over watch expressions. You can see data or other messages displayed, as long as you are in break mode. Another advantage is that the feedback is displayed in a separate area (the Immediate pane), so it does not interfere with output that a user sees. Finally, because you can save this code as part of the workbook, you don't have to redefine these statements the next time you work on the code.

You can evaluate any valid expression in the Immediate pane, including expressions involving properties.

Printing from Code

The **Print** method sends output to the Immediate pane whenever you prefix it with the special Debug object. For example, the following statement prints the value of `Salary` to the Immediate pane every time the statement is executed:

```
Debug.Print "Salary = "; Salary
```

This technique works best when there is a particular place in your code at which the variable (in this case, `Salary`) is known to change. For example, you might put the previous statement in a loop that repeatedly alters `Salary`.

For more information about the Debug object, search the online Visual Basic Reference for *Debug*.

Printing from Within the Immediate Pane

Once you're in break mode, you can use the **Print** method without the Debug object. Type or paste a statement in the Immediate pane, and then press ENTER. The Immediate pane responds by carrying out the statement. In the following example, the lines containing the **Print** method are input you provide in the Immediate pane. The lines immediately following show the responses from Visual Basic.

```
Print Initial
    4
Print Final
    6
```

A question mark (**?**) is useful shorthand for the **Print** method. The question mark means precisely the same thing as **Print** and can be used in any context in which **Print** is used. For example, the statements in the previous sample code could be entered as shown here.

```
? Initial
    4
? Final
    6
```

Executing Commands in the Immediate Pane

You can use the Immediate pane for more than printing the values of variables and properties. In the Immediate pane, you can execute Visual Basic commands to:

- Change the values of variables and properties.
- Run statements that change the behavior of the code.

Visual Basic immediately executes the commands you enter in the Immediate pane.

Assigning Values to Variables and Properties

As you close in on the possible cause of an error, you might want to test the effects of particular data values. In break mode, you can set values with statements in the Immediate pane like these:

```
ActiveWorkbook.Charts(1).AutoScaling = True
MaxTax = .36
```

The first statement alters a property of a chart, and the second assigns a value to a variable.

After you set the values of one or more properties and variables, you can continue execution to see the results. Or you can test the effect on procedures, as described in the next section.

Testing Procedures

The Immediate pane evaluates any valid Visual Basic executable statement, but it doesn't accept data declarations. You can enter calls to procedures, however, which allows you to test the possible effect of either type of procedure with any given set of arguments. Simply enter a statement in the Immediate pane just as you would in a Visual Basic module. For example:

```
X = Quadratic(2, 8, 8)
DisplayGraph 50, Arr1
```

Visual Basic executes the statement and then returns to break mode. At that point, you can see results and test any possible effects on variables or property values.

Note Although all executable statements are supported in the Immediate pane, a control structure is valid only if it can be completely expressed on one line. The following **For** loop is valid in the Immediate pane:

```
For I = 1 To 20 : Print 2 * I : Next I
```

Once you enter break mode, you can execute another procedure from the Immediate pane. By using breakpoints or a **Stop** statement, you can enter break mode a second time. When the second procedure finishes running, either because you stepped all the way through it or because you resumed code execution, Visual Basic returns you to the Debug window and the original break mode.

Using the Immediate Pane More Effectively

Here are some shortcuts you can use in the Immediate pane.

- Once you enter a statement, you can execute it again by moving the insertion point back to that statement and pressing ENTER.
- Before pressing ENTER, you can edit the current statement and alter its effects.
- You can use the mouse or the arrow keys to move around in the Immediate pane. Don't press ENTER unless you are at a statement you want to execute.

Simplifying the Debugging Process

The more accurate your code is in the first place, the less debugging you have to do later. There are a number of things you can do to avoid creating bugs in your code:

- Include numerous comments. As you go back and analyze your code, you'll understand it much better if the purpose of each procedure is stated in comments.
- Develop a consistent naming scheme for the variables in your code. One of the most common sources of errors is incorrectly typing a variable name or confusing one variable with another. For example, you could use a prefix of `gv` to indicate a global, or public, variable. Following this naming scheme tells you that a variable named `gvInitial` is a global variable.

  Another way to avoid confusion over variable names is to use the **Option Explicit** statement. For more information about the **Option Explicit** statement, see "Identifying Variables and Arguments Before Using Them" in Chapter 6.

Tips for Debugging

When you do find errors in your code, there are a number of things you can do to simplify debugging.

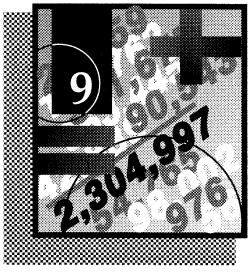
- When your code doesn't produce correct results, browse through the code and try to find the statements that might have caused the problem. Set breakpoints at these statements and restart the code. Breakpoints are described in "Entering Break Mode at a Problem Statement" earlier in this chapter.
- When the program halts, test the values of important variables and properties. Set watch expressions to monitor these values. Use the Debug window to examine variables and expressions. Watch expressions are described in "Monitoring Data with Watch Expressions" earlier in this chapter.

- Single step through your code, using watch expressions to monitor how values change as the code runs.
- If you determine that a variable or property is causing problems in your code, define a watch expression that halts execution when the wrong value is assigned to the variable or property.

This completes your introduction to debugging. You've seen how Visual Basic provides tools that can help you explore procedures as you search for errors in the code. These tools give you the ability to observe the effects of your code variables and properties.

CHAPTER 9

Handling Errors and Error Values



This chapter explains how you handle *run-time errors*—errors that occur while your code is running and that result from attempts to complete an invalid operation. You will also learn about *error values*—errors that procedures return when you call them. For more information about testing and debugging your code, see Chapter 8, "Testing and Debugging Your Code."

Ideally, Visual Basic procedures wouldn't need error-handling code at all. Unfortunately, files can be mistakenly deleted, or you might try to access a property that doesn't exist or set the value of a read-only property, or a procedure can be expected to deal with unexpected values. To handle errors such as these, you need to add error-handling code to your procedures.

Contents

What Happens When You Don't Intercept Errors?

When errors occur in your code and you do not check for them, Visual Basic often generates a run-time error that halts your code, and there's usually nothing the user can do to resume your application. Other errors might not interrupt your code, but they can cause it to act unpredictably.

For example, the following **Function** procedure returns **True** if the specified file exists, and **False** if it does not, but it doesn't contain error-handling code.

```
Function FileExists(filename)
    FileExists = (Dir(filename) <> "")
End Function
```

The **Dir** function returns the first file on your disk that matches the specified filename (given with or without wildcard characters, drive letter or name, or path); it returns a zero-length string if no matching file is found. Therefore `Dir(filename) <> ""` returns **True** if **Dir** returns anything, and **False** if **Dir** returns a zero-length string. The code appears to cover either of the possible outcomes of the **Dir** call.

However, if the drive letter (in Microsoft Windows) or drive name (on the Macintosh) specified in the argument is not a valid drive, the error message `Device unavailable` appears. If the specified drive is a floppy disk drive, this function works correctly only if a disk is in the drive (and if the drive door is closed, if the drive has a door). If not, Visual Basic displays the error message `Disk not ready` and stops your code. Neither of these messages tells you whether the specified file exists, or what you should do next.

If you include error-handling code in procedures, you can often recover from an error and give users the opportunity to complete their task or remedy the situation that caused the error.

Preventing Your Code from Halting or Acting Unpredictably

With the error-handling features in Visual Basic, you can intercept run-time errors, such as those described in the previous section, and take actions to correct them. Intercepting an error is also known as *trapping* an error. You can use the following statements and functions to trap and then handle run-time errors:

- The **On Error GoTo** statement makes an error-handling routine available and specifies the location of the routine within a procedure; it can also be used to disable an error-handling routine.

  ```
  On Error GoTo CheckError
  ```

- The **Err** function returns the number corresponding to the most recent run-time error. It acts like any **Public** variable containing an integer.

  ```
  MsgBox "The most recent error number is " & Err & _
    ". Its message text is: " & Error(Err)
  ```

  For more information about **Public** variables, see "Learning More About Visual Basic Programming" in Chapter 6.

- The **Error** function returns message text corresponding to an error number. Every run-time error has a corresponding error number that identifies it.

  ```
  ErrorMessageText = Error(71)
  ```

Other error handling statements and functions (such as the **Err** and **Error** statements, which are not the same as the **Err** and **Error** functions described here) are described in "Advanced Error-Handling Techniques" later in this chapter. You can also search the online Visual Basic Reference for *Error handling*.

Following is an example of how disk problems, such as an invalid drive or an empty floppy disk drive, can be handled with error-handling code.

```
Function FileExists (filename)
On Error GoTo CheckError      ' Turn on error trapping so error handler
                              ' responds if any error is detected.
    FileExists = (Dir(filename) <> "")
    Exit Function             ' Avoid executing error handler
                              ' if no error occurs.

CheckError:                   ' Branch here if error occurs.
    ' Define constants to represent Visual Basic error code.
    Const ERR_DISKNOTREADY = 71, ERR_DEVICEUNAVAILABLE = 68
    FileExists = False
    If Err = ERR_DISKNOTREADY Then
        Msg = "Put a floppy disk in the drive and close the drive door."
        ' Display message box with an exclamation mark icon and with OK
        ' and Cancel buttons.
        If MsgBox(Msg, vbExclamation + vbOKCancel) = vbOK Then
            Resume
        Else
            Resume Next
        End If
    ElseIf Err = ERR_DEVICEUNAVAILABLE Then
        Msg = "This drive or path does not exist: " & filename
        MsgBox Msg, vbExclamation
        Resume Next
    Else
        Msg = "Unexpected error #" & Str(Err) & ": " & Error(Err)
        ' Display message box with Stop sign icon and OK button.
        MsgBox Msg, vbCritical
        End
    End If
End Function
```

On Error GoTo requires you to use line numbers or line labels that identify where your error-handling code is, as shown in the preceding example. A *line label* follows standard naming conventions and ends with a colon. (`CheckError:` is the line label in the preceding example. Notice that the colon is not used in the **On**

Error GoTo statement.) Each line label must be unique within the module where it is used. It is not case-sensitive and can begin anywhere on a separate line as long as the label is the first nonblank character on the line. (A *line number* works the same way, but you use numbers instead of characters.)

Note In the preceding example, the error numbers that the **Err** function returns are compared to constants that you must define. For a list of these error numbers, search the online Visual Basic Reference for *Trappable errors.*

The constants that begin with the letters "vb" apply to dialog boxes that are displayed by the **MsgBox** function and are built into Visual Basic. For more information about the **MsgBox** function, search the online Visual Basic Reference for *MsgBox.*

When Visual Basic generates the error `Disk not ready` in the preceding example, the code displays a message telling you to choose the OK or Cancel button. If you choose the OK button, the **Resume** statement returns control to the statement at which the error occurred and attempts to run that statement again.

If you choose the Cancel button, the **Resume Next** statement returns control to the statement following the one at which the error occurred (in this case, the **Exit Function** statement).

If the error `Device unavailable` occurs, the code displays a message describing the problem. The **Resume Next** statement then causes the function to continue running at the statement following the one at which the error occurred.

If an unanticipated error occurs, an alternative message is displayed and the code stops at the **End** statement.

Note Using the **End** statement in your own applications can be helpful, as shown in the preceding example. **End** stops execution in an orderly way, closing files, clearing variables, and returning control back to Microsoft Excel. Avoid the **Stop** statement, which stops execution abruptly and can confuse your users.

The Process of Trapping Errors

The example in the preceding section contains an error-handling routine called CheckError. It illustrates three steps that you can apply to most error-handling routines:

- Setting an error trap
- Writing an error-handling routine
- Exiting the routine

Setting an Error Trap

An error trap is enabled when Visual Basic encounters the **On Error** statement, which specifies an *error handler*—the routine within a procedure that handles the error. The error trap remains enabled while the procedure containing it is active—that is, until an **Exit Sub**, **Exit Function**, **End Sub**, or **End Function** statement is run in that procedure. (**Exit Property** and **End Property** are also available for property procedures.)

While only one error trap can be enabled at any one time in any given procedure, you can create several alternative error traps and enable different ones at different times. You can also disable an error trap by using a special case of the **On Error** statement—**On Error GoTo 0**. For more information about disabling error handling, see "Turning Off Error Handling" later in this chapter.

Writing an Error-Handling Routine

The first step in writing an error-handling routine is placing the **On Error GoTo** *line* statement, where *line* is text that indicates the line label identifying the error-handling code.

In the FileExists function that you saw earlier, the label is `CheckError`. A common convention is to place the error-handling code at the end of the procedure, before the **End Function** or **End Sub** statement, with an **Exit Sub** or **Exit Function** statement immediately before the line label. This prevents running the error-handling code if no error occurs.

Use the **Err** function, which returns a numeric code representing the most recent error (including errors you've defined), in combination with the **Select Case** or **If...Then...Else** statement to run specific error-handling code. This is demonstrated in the example in the next section.

For more information about the **Select Case** and **If...Then...Else** statements, see "Running Portions of Code Based on Conditions (Decision Making)" in Chapter 7.

Exiting an Error-Handling Routine

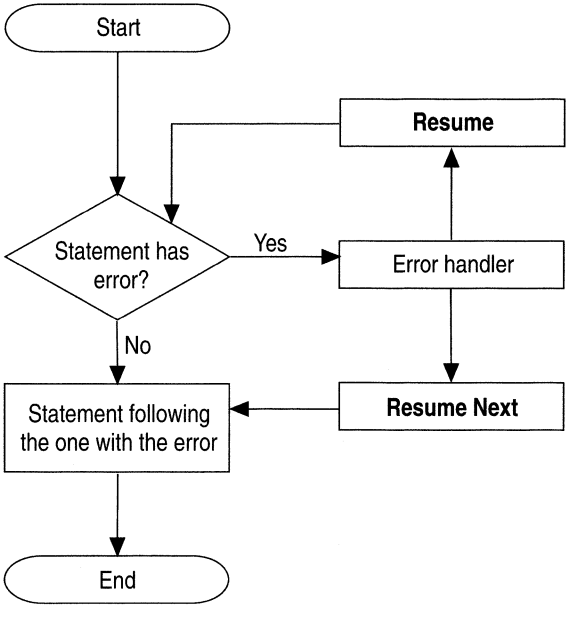
The FileExists function example earlier in this chapter uses the **Resume** statement within the error handler to rerun the statement that originally caused the error, and it uses the **Resume Next** statement to run the statement following the statement at which the error occurred.

You can use these and other statements to exit an error-handling routine depending on the circumstances, as explained in the following table.

Statement	Description
Resume Next	Resumes running the statement immediately following the one that caused the error.
Resume	Resumes running the statement that caused the error. Use it to repeat an operation after correcting the error.
Resume *line*	Resumes running at the label specified by *line*, where *line* is a nonzero line number or line label that must be in the same procedure as the error handler.
Error Err	Triggers the most recent run-time error again. When this statement is run within the error-handling routine, Visual Basic searches the calls list for another error-handling routine.

The preceding table assumes that the error occurred in the same procedure as the error handler. If it did not, then Visual Basic searches the calls list for another error-handling routine. (The calls list is the chain of procedures to which the current procedure returns once it finishes. The calls list is displayed in the Calls dialog box, which you can display by choosing the button to the right of the Procedure box in the Debug window. For more information, see Chapter 8, "Testing and Debugging Your Code.")

The following illustration shows how **Resume** and **Resume Next** differ.



Generally, you use **Resume** whenever the user must make a correction and **Resume Next** whenever the code needs correction, or when you anticipate an error in a loop and you need to restart the operation. However, your error handler can be written so that a particular run-time error is never revealed to the user.

For example, the following **Function** procedure uses error-handling code to perform "safe" division on its arguments without revealing errors that might occur; **Null** is returned if they do occur.

```
Function Divide(numer, denom)
Const ERR_DIV0 = 11, ERR_OVERFLOW = 6, ERR_ILLFUNC = 5
On Error GoTo MathHandler
    Divide  = numer / denom
    Exit Function
MathHandler:
    If Err = ERR_DIV0 Or Err = ERR_OVERFLOW Or Err = ERR_ILLFUNC Then
        Divide = Null        ' If error was Division by zero, Overflow,
                             ' or Illegal function call, return Null.
    Else
        MsgBox "Unanticipated error " & Err & ": " & Error, _
          vbExclamation
        Divide = Null
    End If                   ' In all cases, Resume Next continues
    Resume Next              ' execution at the Exit Function statement.
End Function
```

How Visual Basic Searches for an Error-Handling Routine

When an error occurs within a procedure lacking an enabled error-handling routine, or within an error-handling routine that is currently running, Visual Basic searches the calls list for another enabled error-handling routine.

For example, suppose the following sequence of calls occurs:

1. Procedure A calls Procedure B.
2. Procedure B calls Procedure C.
3. Procedure C calls Procedure D.

If an error occurs in Procedure D and this procedure doesn't have an enabled error handler, Visual Basic searches backward through the pending procedures in the calls list—first Procedure C, then Procedure B, then Procedure A (but no further)—and executes the first enabled error handler it finds. If it doesn't encounter an enabled error handler anywhere in the calls list, it displays the appropriate run-time error message and halts your procedure.

Tip The effect of the search back through the calls list is hard to predict because if a **Resume** or **Resume Next** statement is encountered in the error-handling subroutine, execution continues in the procedure where the error-handling subroutine is found, not necessarily in the procedure where the error occurred. To avoid this, write a fail-safe error-handling procedure that all your error handlers can call as a last resort for errors they cannot handle themselves. This fail-safe procedure can save the user's data and perform an orderly termination of your application.

Turning Off Error Handling

If an error trap has been enabled in a procedure, it is automatically disabled when the procedure finishes running. However, you might want to turn off an error trap earlier. To turn off an enabled error trap, use the **On Error GoTo 0** statement. When Visual Basic runs this statement, errors are detected but not trapped within the procedure. You can use **On Error GoTo 0** to turn off error handling anywhere in a procedure—even within an error-handling routine itself.

For example, try single stepping through a procedure such as this:

```
Sub ErrDemoSub()
On Error GoTo SubHandler
' Error trapping is enabled.
' Errors need to be caught and corrected here.
    Kill "OLDFILE.XYZ"
On Error GoTo 0              ' Error trapping is turned off here.
    Kill "OLDFILE.XYZ"
On Error GoTo SubHandler     ' Error trapping is enabled again.
    Kill "OLDFILE.XYZ"
Exit Sub
SubHandler:                  ' Error-handling routine goes here.
    MsgBox "Caught error."
    Resume Next
End Sub
```

You can also use the following statement to prevent Microsoft Excel from displaying any alert messages.

```
Application.DisplayAlerts = False
```

For more information about alert messages, search the online Visual Basic Reference for *DisplayAlerts*.

Creating Error Values That Don't Interrupt Your Code

The preceding section explains how you handle run-time errors. But there is another, completely different type of error—a user-defined error value. *User-defined error values* are values that you define for your procedures and are always stored in variables of the **Variant** data type.

A common use of this type of error value is in procedures that accept several arguments and return a value. Suppose the return value is valid only if the arguments fall within certain ranges. Your procedure can test the arguments that the user provides and, if they aren't in the acceptable range, you can have the procedure return an appropriate error value.

What an Error Value Is

Variant variables can contain many types of data, including all the intrinsic data types plus **Error**, **Empty**, and **Null**. **Error** is a subtype of the **Variant** type, and when the term "**Error** value" is used, it usually means a variable that is of the **Variant** data type and that contains a value Visual Basic recognizes as a user-defined error. For more information about the **Empty** and **Null** values and the **Error** subtype, see "Other Types of Information Stored in Variant Variables" in Chapter 6.

You use such **Error** values in a procedure to indicate that error conditions have occurred, but unlike normal run-time errors, these errors do not interrupt your code because Visual Basic sees them as ordinary variables. Your procedure can test the **Error** values and take appropriate actions.

How You Create Error Values

Error values are created using the **CVErr** function, whose argument must be an integer or a variable containing an integer.

```
NoRadius = CVErr(2010)

NotANumber = 2020
InvalidArgument = CVErr(NotANumber)
```

The following example uses the **CVErr** function to return an **Error** value. The AreaOfCircle procedure asks for a radius value. The CheckData **Function** procedure checks the `Radius` variable and, if it is not an acceptable number, converts it into one of two possible **Error** values.

```
Public NoRadius, NotANumber

Sub AreaOfCircle()
    Const PI = 3.14
    NoRadius = CVErr(2010)
    NotANumber = CVErr(2020)
    Radius = CheckData(InputBox("Enter the radius"))
    If IsError(Radius) Then
        Select Case Radius
            Case NoRadius
                MsgBox "Error: No radius given."
            Case NotANumber
                MsgBox "Error: Radius isn't a number."
            Case Else
                MsgBox "Unknown error."
        End Select
    Else    ' there's no error
        MsgBox "The area of the circle is " & (PI * Radius ^ 2)
    End If
End Sub

Function CheckData(TheRadius)
    If Not IsNumeric(TheRadius) Then
        CheckData = NotANumber
    ElseIf TheRadius = 0 Then
        CheckData = NoRadius
    Else
        CheckData = TheRadius
    End If
End Function
```

In the AreaOfCircle **Sub** procedure, the **IsError** function returns **True** if its argument is an **Error** value. It is important to remember that a run-time error occurs if you use an error value in an expression in which Visual Basic attempts to compare an error value to a number or other value.

```
If NoRadius = 2010 Then...   'Causes an error. Cannot compare error to a
                             'number or other value.
```

The following code does not cause an error because you can compare error values to other error values.

```
If NoRadius = CVErr(2010) Then...
```

For more information about **IsError** or **CVErr**, search the online Visual Basic Reference for *IsError* or *CVErr*.

Using the Built-in Error Values of Microsoft Excel

Names and cell formulas in Microsoft Excel worksheets can contain one of seven error values (#DIV/0!, #N/A, #NAME?, #NULL!, #NUM!, #REF!, and #VALUE!). You can pass these errors from worksheets to procedures only if the procedures' argument types are **Variant**. To work with these error values, you can test the arguments with the **IsError** function and work with them as if they were user-defined error values, described in the preceding section.

You can also pass error values back to cells in worksheets from a user-defined function, as shown here.

```
Function Commission(SharesSold, PricePerShare)
    If Not (IsNumeric(SharesSold) And IsNumeric(PricePerShare)) Then
        Commission = CVErr(xlErrNum)     ' xlErrNum corresponds to the
                                         ' #NUM! error value.
        Exit Function
    Else
        TotalSalePrice = SharesSold * PricePerShare
        If TotalSalePrice <= 15000 Then
            Commission = 25 + 0.03 * SharesSold
        Else
            Commission = 25 + 0.03 * (0.9 * SharesSold)
        End If
    End If
End Function
```

You work with the Microsoft Excel worksheet error values the same way you work with the user-defined errors—as numbers converted to **Error** values using the **CVErr** function. The only difference is that, for worksheet errors, Visual Basic provides the error numbers as built-in constants, and it also provides literal error values. These items are not provided for user-defined error values. The literal error values must be enclosed in square brackets, as shown in the following table.

Error number (constant)	Literal error value	Converted error value
xlErrDiv0	[#DIV/0!]	CVErr(xlErrDiv0)
xlErrNA	[#N/A]	CVErr(xlErrNA)
xlErrName	[#NAME?]	CVErr(xlErrName)
xlErrNull	[#NULL!]	CVErr(xlErrNull)
xlErrNum	[#NUM!]	CVErr(xlErrNum)
xlErrRef	[#REF!]	CVErr(xlErrRef)
xlErrValue	[#VALUE!]	CVErr(xlErrValue)

If you want to use the literal error value for #NUM! in the preceding example, you substitute [#NUM!] for CVErr(xlErrNum), as shown here.

```
If Not (IsNumeric(SharesSold) And IsNumeric(PricePerShare)) Then
    Commission = [#NUM!]
    Exit Function
.
.
.
```

Advanced Error-Handling Techniques

Visual Basic provides additional functions and statements with which you can construct more sophisticated error-handling code. This section describes the **Error** and **Err** statements (which are not the same as the **Error** and **Err** functions described previously in this chapter), and the following sections demonstrate their use in code.

The **Error** statement simulates an error by actually interrupting your code (unless you have written error-handling code to handle it) with the run-time error specified by the error number you provide. The **Error** statement also sets the value of the **Err** function to the same number you provide. An error number is a whole number in the range 0 to 65,535, the lower range of which is reserved for Visual Basic.

```
Error 71
```

You can also define your own error numbers. For more information about numbering errors, see the note in the next section.

The **Err** statement sets the value that the **Err** function returns.

```
Err = 71
```

If you don't use the **Err** statement, Visual Basic sets the value of the **Err** function based on the most recent run-time error. The **Error** statement also sets the value of the **Err** function, but unlike **Error**, the **Err** statement doesn't simulate an error by attempting to interrupt your code.

Simulating Run-Time Errors

Simulating run-time errors is useful when you are testing your applications, or when you want to treat a particular condition as being equivalent to a run-time error. For example, you might be writing a module that calls routines in a dynamic-link library (DLL) and want the DLL's return values to be handled as actual Visual Basic errors by the rest of your application—that is, to generate a run-time error in Microsoft Excel. (For information about DLLs, see Chapter 10, "Controlling and Communicating with Other Applications.")

You can simulate any Visual Basic run-time error by supplying the error code for that error in the **Error** statement.

```
Error 71     ' Simulate "Disk Not Ready" error.
```

You can also use the **Error** statement to generate your own, user-defined errors by supplying an error code that does not correspond to a Visual Basic run-time error. Of course, this is useful only if you also write code that handles the new errors that you define. For a list of built-in errors, search the online Visual Basic Reference for *Trappable errors.*

Note Microsoft Visual Basic doesn't use all available numbers for its own errors. As new errors are defined in future versions of Visual Basic, the internal error numbers will increase. If you want to generate and trap your own errors, begin your numbering scheme with 50,000 and work your way up.

Minimizing Code Size by Handling Errors in a Central Location

When you add error-handling code to your applications, you'll quickly discover that you're handling the same errors over and over again. With careful planning, you can reduce code size and save effort by writing a few procedures that your error-handling code can call to handle common error situations.

Following is a **Function** procedure called FileErrors that displays a message corresponding to the error that has occurred and, where possible, allows the user to choose a button to specify which action the function should take next. It then returns a code number to the procedure that called it (not shown). The code number indicates which action in a group of actions the program should take.

```
Public Const RESUME_STATEMENT = 0        'Resume
Public Const RESUME_NEXT = 1             'Resume Next
Public Const UNRECOVERABLE = 2           'Unrecoverable error
Public Const UNRECOGNIZED = 3            'Unrecognized error
Public Const ERR_DEVICEUNAVAILABLE = 68
Public Const ERR_BADFILENAMEORNUMBER = 52, ERR_PATHDOESNOTEXIST = 76
Public Const ERR_BADFILEMODE = 54

Function FileErrors (errVal As Integer) As Integer
Dim MsgType As Integer, Msg As String, Response As Integer
    MsgType = vbExclamation
    Select Case errVal
        Case ERR_DEVICEUNAVAILABLE                    ' Error #68
            Msg = "That device appears unavailable."
            MsgType = MsgType + vbAbortRetryIgnore
        Case ERR_BADFILENAMEORNUMBER  ' Errors #64 & 52
            Msg = "That filename is not valid."
            MsgType = MsgType + vbOKCancel
```

```
        Case ERR_PATHDOESNOTEXIST                        ' Error #76
            Msg = "That path doesn't exist."
            MsgType = MsgType + vbOKCancel
        Case ERR_BADFILEMODE                             ' Error #54
            Msg = "Can't open your file for that type of access."
            MsgType = MsgType + vbOKCancel
        Case Else
            FileErrors = UNRECOGNIZED
            Exit Function
    End Select
    Response = MsgBox(Msg, MsgType, "Disk Error")
    Select Case response
        Case vbOK, vbRetry
            FileErrors = RESUME_STATEMENT
        Case vbIgnore
            FileErrors = RESUME_NEXT
        Case vbCancel, vbAbort
            FileErrors = UNRECOVERABLE
        Case Else
            FileErrors = UNRECOGNIZED
    End Select
End Function
```

The preceding **Function** procedure handles a select group of file- and disk-related errors. If the error is not in this group, it returns the value **3**. The procedure that calls this function should then either handle the error itself, call another procedure to handle it, or simply let the original run-time error occur again, as shown in the following example.

```
Function FileOperationsDemo(FileName As String)
'Demo of how you would call the FileErrors Function
On Error GoTo ConfirmFileError
.
.
.
'Code that attempts some file operation
.
.
.
ConfirmFileError:
Action = FileErrors(Err)        'Call the FileErrors function
    Select Case Action
        Case RESUME_STATEMENT   'User chose OK or Retry when FileErrors
                                'displayed its message, so try again.
            Resume
        Case RESUME_NEXT        'User chose Ignore when FileErrors
                                'displayed its message, so ignore error
                                'and continue.
            Resume Next
```

```
        Case UNRECOVERABLE        'User chose Cancel or Abort when
                                  'FileErrors displayed its message,
                                  'so abort procedure.
            Exit Function
        Case Else                 'Cannot handle original run-time
                                  'error, so repeat it and display
                                  'the normal System error.
            Error Err
    End Select
End Function
```

Handling Errors with Inline Code

In previous sections, you've used **On Error GoTo** to jump to an error-handling subroutine and then resume the rest of your code. You can also handle errors without branching to a subroutine.

If you anticipate that a particular statement could cause an error, include an **On Error Resume Next** statement before the potential error-causing statement to prevent interrupting the application. You can then test the value of the **Err** function and act accordingly.

The following procedure is a revision of the FileExists procedure shown earlier in this chapter. This revised version does not branch to a subroutine but handles errors inline.

```
Function FileExists(filename)
Dim Msg

Const ERR_DISKNOTREADY = 71, ERR_DEVICEUNAVAILABLE = 68

' Resets the Err code to zero; handles errors inline
On Error Resume Next

CheckOnFile:
    FileExists = (Dir(filename) <> "")
    If Err = ERR_DISKNOTREADY Then
        Msg = "Put a floppy disk in the drive and close the drive door."
        If MsgBox (Msg, vbExclamation + vbOKCancel) = vbOK Then
            GoTo CheckOnFile
        End If
        FileExists = False
    ElseIf Err = ERR_DEVICEUNAVAILABLE Then
        Msg = "This drive or path does not exist: " & filename
        MsgBox Msg, vbExclamation
        FileExists = False
```

```
    ' If not successful, and if not one of the above errors, handle
    ElseIf Err <> 0 Then
        Msg = "Unexpected error #" & Str(Err) & ": " & Error(Err)
        MsgBox Msg, vbCritical
        FileExists = False
    End If
End Function
```

Handling User Interrupts

When a Visual Basic procedure is running, a user can interrupt the procedure by pressing CTRL+BREAK or ESC (or COMMAND+PERIOD on the Macintosh). While interrupts are useful when you are debugging, you might want to disable them for procedures in finished applications. If you do allow user interrupts in a finished application, however, you can make sure your procedure is notified of them so that it can close files, disconnect from shared resources, or restore modified variables before returning control of the application to the user.

You can trap user interrupts in your procedures by setting the **EnableCancelKey** property to xlErrorHandler. With this property set, all interrupts generate run-time error number 18, which you can trap using an **On Error** statement. You can handle this error to halt the procedure and exit the program. If you use the **Resume** statement to continue after a trapped run-time error, however, the interrupt is ignored.

The following example illustrates a procedure that requires a large period of time to complete. If a user interrupts the procedure, an error is trapped—first to confirm that the procedure should be stopped, and then to exit the procedure in an orderly manner. (Note that you can use only one interrupt handler for each procedure, and the same handler is used for all other run-time errors encountered by that procedure.)

```
Sub ProcessData
    ' Set up user interrupt trapping as a run-time error
    On Error GoTo UserInterrupt
    Application.EnableCancelKey = xlErrorHandler
    'Start a long duration task
    OpenDataFile
    For x = 1 To 1000000
        ProcessRecord x
        WriteOutRecord x
    Next x
    CloseDataFile
    Exit Sub
```

```
UserInterrupt:
    If Err = 18 Then
        If MsgBox ("Stop processing records?" , vbYesNo) = vbNo Then
            ' Continue running at the point we were interrupted
            Resume
        Else
            ' Close open files before returning
            CloseDataFile
            Exit Sub
        End If
    Else
        ' Handle other errors that occur
        MsgBox Error(Err)
        Resume Next
    End If
End Sub
```

You can also ignore user interrupts completely by setting the **EnableCancelKey** property to xlIgnore. In this condition, Microsoft Excel ignores all attempts by the user to interrupt the running procedure.

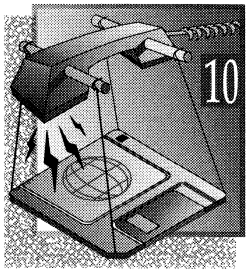
Note Use caution when disabling or ignoring user interrupts. It is possible to write code that never returns or ends. If you disable interrupts by setting the **EnableCancelKey** property to xlIgnore, or if you always use the **Resume** statement to return from a trapped error, your procedure will not return control to the user.

You can restore the default interrupt processing of Microsoft Excel by setting the **EnableCancelKey** property to xlInterrupt. To prevent a procedure from permanently turning off user interrupts, Microsoft Excel always restores the value of the **EnableCancelKey** property to xlInterrupt whenever code execution ends. To ensure that interrupts are handled properly, your procedure must explicitly disable or trap interrupts each time it is run.

This concludes your introduction to the Visual Basic programming language. You are now ready to apply your knowledge to work with other applications; create custom dialog boxes, menus, commands, and automatic and add-in procedures; even work with entire custom applications. The next chapter explains how Microsoft Excel can control and communicate with other applications.

CHAPTER 10

Controlling and Communicating with Other Applications



As you work in Microsoft Excel, you might want to exchange data with other applications. For example, you might want to create a Microsoft Word document from data on a worksheet. Using Visual Basic, you can automate this simple scenario and much more. You can control other applications from within Microsoft Excel—you can even create sophisticated custom applications that integrate Microsoft Excel, Microsoft Word, Microsoft Project, and other applications.

Because there are many types of applications, there are also various ways to work with those applications from Visual Basic. In this chapter, you will learn the different ways you can use Visual Basic to communicate with and carry out commands in other applications. This chapter provides an overview of these ways and then discusses each way in turn.

Contents

Ways to Work with Other Applications

Visual Basic provides four ways to work with other applications:

- Object linking and embedding (OLE) Automation
- Linked and embedded objects
- Dynamic data exchange (DDE)
- Sending keystrokes

OLE Automation is a way to control other applications from Visual Basic. Applications that support OLE Automation provide objects that you use in the same way that you use Microsoft Excel objects. You can use Visual Basic with applications that support OLE Automation to perform these tasks:

- Opening, creating, and saving documents
- Inserting, deleting, or retrieving data
- Carrying out menu or macro commands

Linking and *embedding* are ways to display data from another application in Microsoft Excel. For example, you use linking and embedding to include graphics, formatted text, sounds, or other special kinds of data in a worksheet. You can use Visual Basic to insert, update, and modify linked and embedded objects.

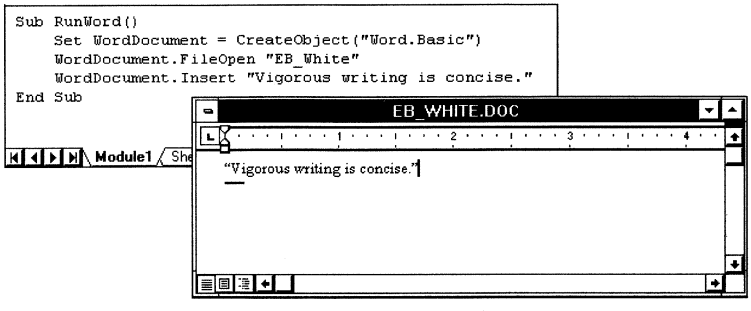
DDE can exchange data with or send commands to applications that don't support OLE Automation or embedding. However, as more applications provide objects for OLE Automation, linking, and embedding, DDE will be used less.

Finally, sending keystrokes is handy when an application doesn't provide any other way to communicate.

Note Macintosh-based applications do not support sending keystrokes.

Using OLE Automation with Applications

Automation can be used to start applications and create objects. The objects that are created through Automation do not necessarily appear as part of a worksheet. Instead, they are usually displayed by the application in which they are created.



OLE Automation is especially useful if you need to use features of one application, such as Microsoft Word, in another application, such as Microsoft Excel. With OLE Automation, you can integrate features from both applications in a single procedure.

▶ **To use OLE Automation**

1. Create a new object, or get an existing object.
2. Write a procedure that uses the object's methods and properties.
3. Close the object when you are finished.

These steps are described in detail in the next sections.

Creating and Getting Objects

When you create new objects or get existing ones, Visual Basic starts the source application for the object, if it is not already running. However, the application does not become active if it is not already active.

How you create or get an object depends on whether the application provides an object library. Most applications, such as Microsoft Project, provide object libraries.

To	Do this
Create a new object or get an existing object from an application that provides an object library	Refer directly to the object, or use the **Set** statement to create a variable that refers to the object.
Create a new object from an application that does not provide an object library	Use the **CreateObject** statement to create a variable that refers to the object.
Get an existing object from an application that does not provide an object library	Use the **GetObject** statement to create a variable that refers to the object.

For more information about determining whether an application provides an object library, see "Adding References to Applications and Workbooks" later in this chapter.

Referring Directly to Objects

Referring to an object directly is useful in many situations, such as when you want to perform a single action on an object. You can refer directly to an object in another application the same way that you refer to objects in Microsoft Excel.

The following example starts Microsoft Project and sets its **WindowState** property to maximized.

```
MSProject.WindowState = pjMaximized
```

Creating a Variable for Frequently Used Objects

If you want to use the same object repeatedly, use the **Set** statement to create a variable to refer to that object. After you create a variable, you can use the shorter variable name when performing actions on the object.

To create a variable that refers to a new object, use the following syntax.

Set *objectvariable* = *application.objectname*

The following example starts Microsoft Project with a new project and establishes the variable `MyProject` to refer to the new project.

```
Set MyProject = MSProject.ActiveProject
MyProject.Manager = "Me"     ' Assign the Manager property.
```

The word `MSProject` is the name of the application as it is registered with Visual Basic. The **ActiveProject** property identifies the project referred to.

To create a variable that refers to an existing object, use the following syntax.

Set *objectvariable* = *application.objectname*(*index*)

The following example starts Microsoft Project and establishes the variable `ThisProject` to refer to the first open project.

```
Set ThisProject = MSProject.Projects(1)
```

What you use as the *index* argument is determined by the object you want to retrieve. For example, the Range object in Microsoft Excel takes a range of cells.

```
Set MyRange = Range("A1","C5")
```

In the preceding example, you can omit the application name (`Excel`) because there are no other objects with the same name. To find out whether you need to include the application name when using the **Set** statement, see "How Visual Basic Searches References" later in this chapter.

Creating a New Object in an Application

If an application is not registered with Visual Basic, you must use the **CreateObject** function to create a new object in that application. Use the following syntax.

Set *objectvariable* = **CreateObject(class :=** *programmaticID*)

The following example starts Microsoft Word and establishes the variable `MSWord` to refer to that application.

```
Set MSWord = CreateObject(class := "Word.Basic")
```

The argument "Word.Basic" is the *programmatic identifier* for Microsoft Word. Programmatic identifiers are registered in your system when you install an application.

Getting an Existing Object from an Application

Use the **GetObject** function to get an existing object from an application. The object can be a file on a disk or an object from an application that is running.

To get an object from a running application, use the following syntax.

Set *objectvariable* = **GetObject(class :=** *programmaticID*)

When working with Microsoft Word version 6.0 or earlier, you cannot use named arguments with **GetObject**. The following example establishes the variable ActiveMSWord to refer to a running instance of Microsoft Word.

```
Set ActiveMSWord = GetObject("", "Word.Basic")
```

Just as with **CreateObject**, the argument "Word.Basic" is the programmatic identifier for Microsoft Word. Unlike **CreateObject**, however, **GetObject** does not start Word if it is not already running. Also, if multiple instances of Word are running, you cannot predict which instance ActiveMSWord will refer to.

You can use this form of the **GetObject** function in conjunction with the **CreateObject** function. For example:

```
Function GetAnObject(ProgrammaticID)
    On Error Resume Next
    Set GetAnObject = GetObject("", ProgrammaticID)
    If Err Then Set GetAnObject = CreateObject(class:=ProgrammaticID)
    If Err Then
        MsgBox "Couldn't create object"
        GetAnObject = False
    End If
End Function
```

In the preceding example, the GetAnObject function uses **GetObject** to return an object from a running application. If the application isn't running, GetAnObject starts the application using **CreateObject**. If that doesn't work, then the function displays a message and returns **False**.

To get an object from a file, use the following syntax.

Set *objectvariable* = **GetObject(***filename*, *programmaticID*)

The following example starts Microsoft Project version 4.0, if it is not already running, and opens the file MYPROJ.MPP.

```
Set myObj = GetObject("c:\winproj\myproj.mpp", "MSProject.Project.4")
```

Having set the variable `myObj`, you can use `myObj` to refer to the project in subsequent code.

You cannot open files in this manner in all applications. For instance, you can't do this with Microsoft Word version 6.0, because the WordBasic programming language relies on selection rather than objects. To select a file in Word, use the **CreateObject** function to start Word, and then open a file using the WordBasic **FileOpen** method. For example:

```
Set MSWord = CreateObject("Word.Basic")      ' Starts a copy of Word.
MSWord.FileOpen name := "EB_White"           ' Opens the file EB_White.
```

There is an important difference between the preceding two examples. The variable `MSWord` refers to a running instance of Word. In contrast, the variable `myObj` refers to a specific file. In the Word example, your code acts on the active document. If the active document changes in Word, any code you write will act on the new active document. In the Microsoft Project example, your code affects a particular file, specified by the variable `myObj`, regardless of which document is active.

For information about whether you can get a file for automation from a specific application, see that application's documentation.

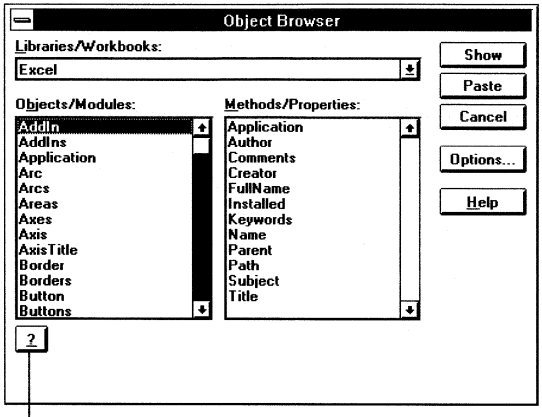
Viewing Applications and Objects

With the Object Browser dialog box, you can view the applications and objects installed in your system. The Object Browser dialog box also displays any procedures that are included in open workbooks or add-in applications in Microsoft Excel. For more information about using procedures from other workbooks and add-in applications, see Chapter 13, "Creating Automatic Procedures and Add-in Applications."

▶ **To view the applications installed in your system and their objects**

1. Switch to a Visual Basic module.
2. From the View menu, choose Object Browser.
3. In the Libraries/Workbooks box, type or select the name of the application you want to view.

   Visual Basic displays the application's objects in the Objects/Modules box.



Displays complete online reference information for the selected item

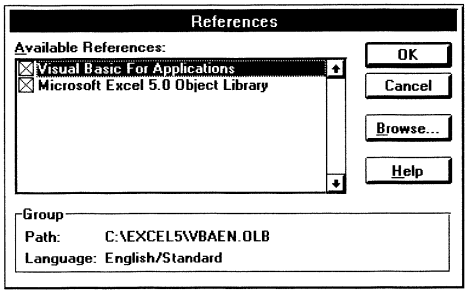
Adding References to Applications and Workbooks

In order for Visual Basic to view the items provided by another application, you need to establish a reference to that application. After you establish a reference, the application's objects are available to Visual Basic and are displayed in the Object Browser dialog box.

▶ **To add a reference to an application or workbook**

1. Switch to a Visual Basic module.
2. From the Tools menu, choose References.
3. In the Available References box, select the name of the application or workbook you want access to.

   If an application does not provide an object library, its name does not appear in this box.



Tip Limiting the number of references speeds up compilation, because Visual Basic searches each of the selected applications in order to resolve symbol names.

For more information about viewing the objects provided by an application or workbook, see the preceding section, "Viewing Applications and Objects."

How Visual Basic Searches References

When you refer to an object in code, Visual Basic searches each application that is selected in the References dialog box in the order that the applications are displayed. If two applications use the same name for an object, Visual Basic uses the first name that it finds. This principle also applies to the methods, properties, and predefined constants displayed in the Object Browser dialog box.

In general, it is a good idea to be as specific as possible when referring to objects defined by other applications or workbooks. For example:

```
' Fully qualified object name sets the Microsoft Project window caption.
MSProject.Caption = "Project Caption"

' Unqualified name causes Visual Basic to use the first object
' it finds with the name Caption -- in this case, Microsoft Excel.
Caption = "Microsoft Excel Caption"
```

Using an Application's Objects

After you have created a variable that refers to an OLE Automation object, the object can be manipulated in Visual Basic in the same way as any Microsoft Excel object. You use the *object.property* syntax to set and return the object's property values or to perform methods on the object.

Some objects contain other objects. For example, a Task object is contained in a project object of Microsoft Project. You can include multiple objects, properties, and methods in the same line of code using the dot syntax, just as you do with Microsoft Excel objects. For example:

```
MSProject.ActiveProject.Tasks(1).Start = Now
```

Some applications, such as Microsoft Word, provide existing macro language statements as methods. These methods take the arguments and use the syntax described in the application's macro language documentation. For example:

```
' Get running instance of Word.
Set MyWord = GetObject("", "Word.Basic")

' Save all open documents (same as WordBasic FileSaveAll statement).
MyWord.FileSaveAll
```

Quitting an Application

Many OLE Automation objects support a method that closes the object and quits the application that created it. Because OLE Automation objects can use a significant amount of memory in your procedure, it is a good idea to explicitly close an object when you no longer need it. Use the appropriate method to close an object. For example:

```
' Closes the Window object.
Window.Close
' Quits the application that created the object.
MyApplication.FileExit
```

When an object has been closed, either by your procedure or by the user, any variables that refer to the object no longer refer to valid objects. For example:

```
Set MyApplication = CreateObject("ApplicationXYZ")
MyApplication.FileNew
MyApplication.Insert "Some text"
MyApplication.FileExit
MyApplication.StartOfDocument    ' Causes an error!
```

Handling Run-Time Errors

In the preceding example, `MyApplication.FileExit` quits another application, so the subsequent statement causes an error. Because users and system errors can quit applications unpredictably, it is a good idea to check for this error when working with objects from other applications. For example:

```
Function SearchFiles(SearchText, FileList())
    On Error Goto ErrorTrap
    Set MyWord = CreateObject("Word.Basic")
    For Each Name In FileList
        MyWord.FileOpen Name
        MyWord.EditFind SearchText
        If MyWord.EditFindFound() Then
            FoundList = FoundList & ", " & Name
        End If
        MyWord.FileClose
    Next
    SearchFiles = FoundList
    Exit Function

ErrorTrap:
    Select Case Err
        Case 440                ' OLE Automation error.
            Tries = Tries + 1
            ' Make up to 5 attempts to restart Word.
            If Tries < 5 Then
                Set MyWord = CreateObject("Word.Basic")
                Resume
            Else
                MsgBox "Couldn't restart Word."
                Stop
            End If
        Case Else
            MsgBox "The following error occurred:" & Err
            Stop
    End Select
End Function
```

If any error other than error 440 occurs in the preceding example, the procedure displays the error and stops. The application that provides the object might pass back its own error, such as WordBasic error 1078, which indicates that the document does not exist. In some cases, an application might use the same error code that Visual Basic uses for a different error. In these cases, you should use **On Error Resume Next** and check for errors immediately after each line that might cause an error. This type of error checking is called *inline error handling*.

For more information about the errors that an application might pass back, see that application's documentation. For more information about trapping errors, see Chapter 9, "Handling Errors and Error Values."

Note When a variable loses scope, the object and its application are not closed. However, you can no longer use that variable to refer to the object. If the object is still active, you can use the **GetObject** function to assign another variable to the object.

Using Nonstandard Names

Some applications might name objects, properties, or methods using characters that have special meaning in Visual Basic. For example, Microsoft Word provides methods that return strings. The name of each of these methods ends in a dollar sign ($), which has special meaning in Visual Basic. To use one of the Microsoft Word string functions in Visual Basic, enclose the name in square brackets.

```
NameofStyle = MyWord.[StyleName$]()
```

Using OLE Automation with DLLs

Dynamic-link libraries (DLLs) and *code resources* contain procedures that you can use much like Visual Basic functions and statements. DLLs and code resources can considerably extend what you are able to do within a procedure.

Note Though there are technical differences between DLLs and Macintosh code resources, they are nearly identical in use. For brevity, this section uses the term *DLL* to refer to dynamic link-libraries and code resources.

Some DLLs support OLE Automation, which makes them easier to use with Visual Basic. DLLs that support automation appear in the References dialog box, and their procedures appear in the Object Browser dialog box.

▶ **To use a procedure from a DLL**

1. If the DLL supports OLE Automation, select the DLL in the References dialog box.

   For instructions, see "Adding References to Applications and Workbooks" earlier in this chapter.

   –Or–

   If the DLL does not support OLE Automation, declare the DLL.

   For instructions, see the following section, "Declaring DLLs."

2. Call the procedure.

   For instructions, see "Calling DLL Procedures" later in this chapter.

Important Visual Basic can't verify that you are passing correct values to a DLL. If you pass incorrect values, the procedure might fail, which might cause your Visual Basic procedure or application to crash. If you have not saved changes to your workbook when this occurs, you might lose some work, and you might have to restart your operating system.

Declaring DLLs

If a DLL does not support OLE Automation, use the **Declare** statement to give Visual Basic the information it needs to find and run the DLL procedures. After you have declared a DLL procedure, you can use it in your code just as you do any other procedure.

To declare a DLL procedure, place a **Declare** statement at the beginning of a module, using the following syntax.

Declare Function *globalname* **Lib** "*libname*" [**Alias** "*alias*"] (*arguments*)

–Or–

Declare Sub *globalname* **Lib** "*libname*" [**Alias** "*alias*"] (*arguments*)

For example:

```
' Declare Windows DLL.
Declare Function GetSystemMetrics Lib "User" (ByVal n As Integer) _
    As Integer

' Declare Macintosh code resource.
Declare Sub MessageAlert Lib "MyHd:MyAlert" Alias "XLVB$MyAlert" _
    (ByVal N As Integer)
```

The **Lib** "*libname*" clause tells Visual Basic where to find the DLL. If the DLL is in your system's path, you can omit the directory and drive name.

The **Alias** "*alias*" clause indicates that the procedure has a different name in the DLL than in your Visual Basic code. Use this clause if the procedure has a name that is not valid in Visual Basic.

The *arguments* placeholder is a list of arguments used when the DLL procedure is called. The following considerations are important when you specify the arguments for a DLL procedure declaration:

- By default, Visual Basic passes all arguments by reference. However, many DLL procedures require an argument to be passed by value. To pass an argument by value, place the **ByVal** keyword in front of the argument declaration in the **Declare** statement.
- Some DLL procedures can accept more than one type of data for the same argument. To pass more than one type of data, declare the argument with the **As Any** keywords to remove type restrictions.
- When passing strings, you must use the **ByVal** keyword to convert a Visual Basic string into a null-terminated string.

The following table lists common DLL arguments and their Visual Basic equivalents.

DLL argument	In Visual Basic, declare as	Call with
Boolean	*I* **As Integer**	Any **Integer** or **Variant** variable
Pointer to a string (`LPSTR`)	**ByVal** *S* **As String**	Any **String** or **Variant** variable
Pointer to an integer (`LPINT`)	*I* **As Integer**	Any **Integer** or **Variant** variable
Pointer to a long integer (`LPDWORD`)	*L* **As Long**	Any **Long** or **Variant** variable
Pointer to a structure (for example, `LPRECT`)	*S* **As** *Rect*	Any variable of that user-defined type
Integer (`INT`, `UINT`, `WORD`, `BOOL`)	**ByVal** *I* **As Integer**	Any **Integer** or **Variant** variable
Handle (`hWnd`, `hDC`, `hMenu`, and so on)	**ByVal** *h* **As Integer**	Any **Integer** or **Variant** variable
Handle (Windows NT™)	**ByVal** *h* **As Long**	Any **Long** or **Variant** variable
Long (`DWORD`, `LONG`)	**ByVal** *L* **As Long**	Any **Long** or **Variant** variable
Pointer to an array of integers	*I* **As Integer**	The first element of the array, such as `I(0)`

DLL argument	In Visual Basic, declare as	Call with
Pointer to a void (`void *`)	**As Any**	Any variable (use **ByVal** when passing a string)
Void (function return value)	**Sub** procedure	Not applicable
`NULL`	**As Any**	**ByVal 0&**

If the DLL uses the C language calling convention, rather than the Pascal calling convention, include the **CDecl** option when declaring a DLL procedure. For more information about calling conventions and declaring DLLs, search the online Visual Basic Reference for *Declare* and *Call.*

Calling DLL Procedures

After you have selected a DLL in the References dialog box or declared the procedure using the **Declare** statement, you can call the DLL procedure just as you do a Visual Basic statement or function. For example:

```
' Call Windows DLL.
Const SM_MOUSEPRESENT = 19
If GetSystemMetrics(SM_MOUSEPRESENT) Then MsgBox "Mouse installed"

' Call Macintosh code resource.
MessageAlert 0
```

If a DLL does not support OLE Automation, then:

- String arguments must be declared with the **ByVal** keyword.
- DLLs cannot return strings.
- DLLs cannot increase the length of strings passed to them. You can avoid this problem by making the string argument long enough that the DLL procedure does not write past the end of it. For example:

  ```
  Dim StringArgument As String * 255
  ```

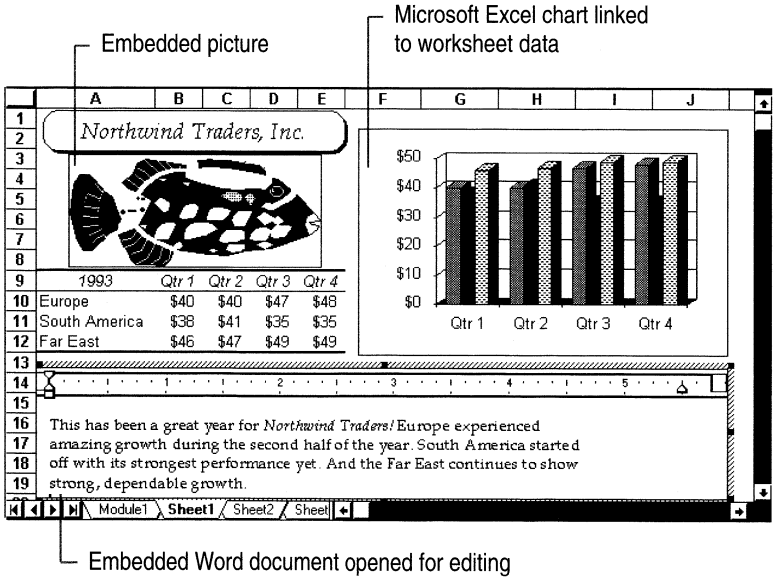
- DLL procedures cannot access arrays.
- User-defined types cannot be passed by value.
- User-defined types passed to a DLL procedure cannot contain variable-length strings.
- Properties must be passed by value.
- You cannot pass an object, such as a worksheet, to a DLL procedure.

In some situations, DLLs use special kinds of arguments:

- When the DLL procedure calls for a memory buffer, use strings. Make sure that the string is long enough to accept whatever data the procedure supplies.
- To pass a null pointer to a procedure, declare the argument with the **As Any** keywords and pass the expression **ByVal 0&**. You can use this technique to pass a null pointer to any procedure that accepts an argument by reference.
- When a procedure takes a handle as an argument, always declare it as a **ByVal Integer** or **ByVal Long**. DLL functions that return a handle can be declared as **Integer** or **Long** functions. Handles are identification numbers, not pointers or numeric values. You cannot perform mathematical operations on them.

Using Linked and Embedded Objects

Applications that support linking and embedding can insert objects in a Microsoft Excel worksheet. For example, a Microsoft Draw graphic can be embedded in a Microsoft Excel worksheet. Linked and embedded objects appear as part of a worksheet—they can be moved, deleted, or edited in the same manner as any other part of the worksheet.



The difference between linked and embedded objects is where their data is stored. Data associated with a linked object is stored in its original document by the application that created it. Data associated with an embedded object is stored in the file in which the object is embedded. An object that is embedded in a Microsoft Excel workbook is saved with the workbook.

The following sections describe how to use linked and embedded objects with the Visual Basic language. For more information about linking and embedding objects using the Microsoft Excel menus and dialog boxes, see Chapter 41, "Sharing Data and Graphics with Other Applications" in the *Microsoft Excel User's Guide.*

Visual Basic provides two objects, OLEObjects and OLEObject, to insert, retrieve, and manipulate linked and embedded objects. The following methods apply to linked and embedded objects.

Method	Purpose
Delete	Deletes a linked or embedded object.
Update	Updates a linked object from its source application.
Verb *OLEverb*	Performs an action on the object. These actions are defined by the application that provides the object. Most linked or embedded objects support the xlPrimary and xlOpen verbs.

The following properties apply to linked and embedded objects.

Property	Purpose
Object	Returns the automation object if the object supports automation.
OLEType	Returns **xlOLEEmbed** if the object is embedded, or **xlOLELink** if the object is linked.
AutoUpdate	Determines whether or not a linked object is automatically updated when the source changes.

For more information about an object, method, or property, search the online Visual Basic Reference for the name of the item.

▶ **To use linked or embedded objects from Visual Basic**

1. Get an existing linked or embedded object, or insert a new one.
2. Perform tasks with the object, using methods and properties.

For example, the following procedure updates all of the linked objects on a worksheet.

```
Sub UpdateAllLinks()
    ' Get each linked or embedded object in the active worksheet.
    For Each LinkedObject in ActiveSheet.OLEObjects
        ' If the object is linked, then update it from its source.
        If LinkedObject.OLEType = xlOLELink Then LinkedObject.Update
    Next
End Sub
```

You can use similar code to perform various tasks on linked and embedded objects selectively. The following sections provide more detail about getting, inserting, and performing specific tasks on linked and embedded objects.

Getting Existing Linked and Embedded Objects

To insert a linked or embedded object in a worksheet using Visual Basic, use the following syntax.

sheet.**OLEObjects**(*item*)

OLEObjects can be used with the **Set** statement to create a variable that refers to a specific linked or embedded object.

In the following example, `MyObject` refers to the first linked or embedded object in the worksheet Sheet1.

```
Set MyObject = Worksheets("Sheet1").OLEObjects(1)
```

If the source application for a linked or embedded object supports OLE Automation, you can find out much more about the content of the object. For more information, see "Using Linked and Embedded Objects with OLE Automation" later in this chapter.

Inserting an Embedded Object

To insert a new embedded object, use the following syntax.

sheet.**OLEObjects.Add** *classtype*

For example:

```
' Embeds a new drawing.
Set EmbeddedDrawing1 = ActiveSheet.OLEObjects.Add (classtype:="MSDraw")
```

To insert a embedded object from an existing file, use the following syntax.

sheet.**OLEObjects.Add filename :=** *filename*, **link := False**

For example:

```
' Embeds a drawing from an existing file.
Set EmbeddedDrawing2 = ActiveSheet.OLEObjects.Add _
    (filename:="guernica", link := False)
```

Because embedded objects are stored with the workbook, changes to the file GUERNICA in the preceding example are not reflected in the embedded object. If you don't provide a filename, Visual Basic assumes that the object is embedded and ignores the **link** argument.

Inserting a Linked Object

To insert a linked object using Visual Basic, use the following syntax.

sheet.**OLEObjects.Add filename :=** *filename*, **link := True**

For example:

```
' Inserts a linked drawing.
Set LinkedDrawing = ActiveSheet.OLEObjects.Add (filename:="guernica", _
    link := True)
```

Because linked objects are stored in the original file, you must provide a filename when you insert the object. Changes to the file GUERNICA in the preceding example are reflected in the linked object when you update the object.

For example:

```
' Updates the linked object immediately.
LinkedDrawing.Update
```

Performing Actions on Linked and Embedded Objects

Using Visual Basic, you can perform actions on linked and embedded objects, such as activating the object for editing. Use the following syntax.

OLEobject.**Verb** *OLEverb*

For example:

```
' Perform the default action on each linked or embedded object.
For Each MyObject In ActiveSheet.OLEObjects
    MyObject.Verb xlPrimary
Next MyObject
```

Many linked and embedded objects provide actions that are unique to that object. For example, an embedded sound recording might provide a Play action. To find out what actions a linked or embedded object provides, see the documentation for the object's source application. Actions may be documented as OLE verbs.

Tip Most applications support the xlOpen and xlPrimary verbs. The xlPrimary verb causes the default action for the object. The default action is usually opening the object for editing, but it is defined by the application providing the object and can be another action, such as playing a sound recording.

Editing Linked and Embedded Objects

You can edit some embedded objects in place, depending on their source applications. *In-place editing* means that you activate the object you want to edit and then change it directly on the worksheet, rather than opening it in a new window.

If the object is linked or if its source application does not support in-place editing, you must open the object in a new window in order to edit it.

To activate an embedded object for in-place editing, use the following syntax.

OLEobject.**Activate**

For example:

```
' Insert an embedded drawing.
Set EmbeddedDrawing = ActiveSheet.OLEObjects.Add _
    (filename:="guernica", link := False)
' Activate the object.
EmbeddedDrawing.Activate
```

To open a linked or embedded object in a new window for editing, use the following syntax.

OLEobject.**Verb xlOpen**

For example:

```
' Open the source application for editing.
EmbeddedDrawing.Verb xlOpen
```

Updating Linked Objects

To update a linked object using Visual Basic, use the following syntax.

OLEobject.**Update**

For example:

```
' Update the link from the source.
LinkedDrawing.Update
```

Updating is only meaningful for linked objects. Embedded objects are stored with the document in which they are embedded and are therefore always up-to-date.

Deleting Linked and Embedded Objects

To delete a linked or embedded object using Visual Basic, use the following syntax.

OLEobject.**Delete**

For example:

```
' Delete the link or embedded object.
LinkedDrawing.Delete
```

Using Linked and Embedded Objects with OLE Automation

If an application supports OLE Automation, you can use that application's properties and methods with the objects it provides for linking and embedding. For example, you can use Microsoft Word methods on Word documents embedded in a worksheet.

▶ **To use a linked or embedded object with OLE Automation**

1. Use the linked or embedded object's **Object** property to refer directly to the object or to create a variable that refers to the object.

   The OLEObject **Object** property has the following syntax.

   OLEobject.**Object**

2. Write code to use the properties and methods provided by that application for the object.

For example:

```
' Create a new embedded object.
Set MyEmbeddedWordDocument = ActiveSheet.OLEObjects.Add (classtype := _
    "word.document.6")
' Create a variable that refers to Word using the embedded object.
Set MyWordBasic = MyEmbeddedWordDocument.Object.Application.WordBasic
MyEmbeddedWordDocument.Activate
' Insert some text using the Insert method of Word.
MyWordBasic.Insert "Insert this text in the embedded document."
```

You can use a linked or embedded object's properties and methods to retrieve information about the content of the objects on a worksheet.

The following example checks the linked and embedded objects in the active worksheet and lists the Word documents that include the name "Dorian Eliot."

```
For Each MyObject In ActiveSheet.OLEObjects
    ' Assume all OLE objects are from Word
    ' Activate the object because Word requires the selection
    MyObject.Activate
    With MyObject.Object.Application.WordBasic
        ' Search for a name.
        .StartOfDocument
        .EditFind Find:="Dorian Eliot", Direction:=0
        ' If the name is found, add it to a list.
        If .EditFindFound() = -1 Then
            FileList = FileList & ", " & .Filename()
        End If
    End With
    ' Deactivate the OLE object in preparation for the next object
    Range("A1").Select
Next
```

Using DDE

Dynamic data exchange (DDE) is a mechanism that permits two applications to "talk" to each other by continuously and automatically exchanging data. DDE automates the manual cutting and pasting of data between applications.

The following table lists the tasks that Microsoft Excel performs with DDE and the method used for each task.

Task	Method
Starting DDE	**DDEInitiate**
Getting text from another application	**DDERequest**
Sending text to another application	**DDEPoke**
Carrying out a command in another application	**DDEExecute**
Ending DDE	**DDETerminate**

Note Not all applications support DDE. To find out whether an application supports DDE, see the documentation for the application.

Starting DDE

Before you can perform any other tasks with DDE, you must establish a connection with the other application. This connection is called a *DDE channel*. The DDE channel is used as an argument in all of the subsequent DDE methods.

▶ **To start DDE and establish a DDE channel**

1. Start the other application, if it is not already running.
2. Use the **DDEInitiate** method to establish a connection and return the DDE channel.

   The **DDEInitiate** method has the following syntax.

   channelNumber = **DDEInitiate**(*applicationname*, *topicname*)

The following example starts Microsoft Word and establishes a DDE channel to the file EB_WHITE. The variable `WordChannel` can now be used by the other DDE methods.

In Microsoft Excel for Windows, use the following code.

```
Shell "WINWORD.EXE C:\DOCS\EB_WHITE.DOC"
WordChannel = DDEInitiate("WinWord", "C:\Docs\EB_White.DOC")
```

In Microsoft Excel for the Macintosh, use this code.

```
Shell "Microsoft Word"
WordChannel = DDEInitiate("Microsoft Word", "HD1:Documents:EB_White")
```

The following table gives the DDE names for some Microsoft applications.

Application	DDE application name
Microsoft Word for Windows	WinWord
Microsoft Word for the Macintosh	Microsoft Word
Microsoft Project for Windows	Project
Microsoft Project for the Macintosh	Microsoft Project
Microsoft Access®	MSAccess
Microsoft FoxPro® for Windows	FoxPro
Microsoft Windows Program Manager	ProgMan

Getting Text and Numbers

After you start DDE using the **DDEInitiate** method, use the **DDERequest** method to get text and numbers from the DDE channel. Use the following syntax.

variable = **DDERequest**(*channelNumber*, *item*)

Variable must be of the **Variant** data type because DDERequest always returns an array.

The following example assigns the text at the bookmark `DDE_Link1` in a Word document opened for DDE.

```
WordNote1 = DDERequest(WordChannel, "DDE_Link1")
```

DDERequest only returns text; it cannot be used to return pictures or formatting in a document.

Sending Text and Numbers

After you start DDE using the **DDEInitiate** method, use the **DDEPoke** method to send text and numbers to the DDE channel. Use the following syntax.

DDEPoke(*channelnumber*, *item*, *data*)

The following example replaces the text at the bookmark `DDE_Link1` in a Word document with the value of cell A1 on the active sheet.

```
DDEPoke WordChannel, "DDE_Link1", Cells(1, 1)
```

DDEPoke only sends text or numbers; it cannot insert a picture or formatting in a document.

Sending Commands

After you start DDE using the **DDEInitiate** method, use the **DDEExecute** method to carry out a command in another application through DDE. Use the following syntax.

DDEExecute(*channelNumber*, *DDEcommand*)

The following example opens a new document in Word.

```
DDEExecute WordChannel, "[FileOpen .Name = ""NEW.DOC""]"
```

The commands that you can send through DDE are determined by the other application. Microsoft Word provides all of the WordBasic commands through DDE, although using OLE Automation is more straightforward. For information about sending WordBasic commands to Microsoft Word using OLE Automation, see "Using OLE Automation with Applications" earlier in this chapter.

Ending DDE

Use the **DDETerminate** method to end DDE with another application. After you use **DDETerminate**, the DDE channel cannot be used unless you start DDE again with the **DDEInitialize** method.

To end DDE, use the following syntax.

DDETerminate(*channelnumber*)

The following example ends the DDE session.

```
DDETerminate WordChannel
```

It is a good idea to end DDE as soon as you are finished with the task at hand. You can have a limited number of channels open at once, and they remain open until you close them with the **DDETerminate** method.

Working with DDE Links

Use the information returned from the **LinkSources** method to update or modify Microsoft Excel and DDE links in a workbook. Use the following syntax.

linkarray = *workbook*.**LinkSources**(*type*)

The *type* argument identifies the type of link returned. **LinkSources** can return four types of links.

Type of link	Description
xlExcelLinks (default)	Returns a list of the Microsoft Excel links in the workbook
xlOLELinks	Returns a list of the OLE and DDE links in the workbook
xlPublishers, xlSubscribers	Return a list of the editions and references of the edition (Macintosh only)

The following table lists other workbook methods that apply to links.

Method	Purpose
LinkInfo	Returns information about a link
OpenLinks	Opens the source document for editing
SetLinkOnData	Sets up a procedure to run whenever the data in a link changes
UpdateLink	Updates a link

The following example opens each of the Microsoft Word links in a workbook for editing, then updates all the Microsoft Excel links in the workbook.

```
LinkArray = ActiveWorkbook.LinkSources(xlOLELinks)
For LinkItem = 1 to UBound(LinkArray)
    If LinkArray(LinkItem) Like "*Word*" Then
        ActiveWorkBook.OpenLinks(LinkArray(LinkItem), type:=xlOLELinks)
    End If
```

```
Next LinkItem
' Update the OLE and DDE links to reflect changes.
ActiveWorkbook.UpdateLink type:=xlExcelLinks
```

For more information about the workbook methods that apply to links, search the online Visual Basic Reference for the name of each method.

Handling Errors

When you use DDE, you need to plan for unexpected events that might occur in the application with which you are exchanging data. For example, a user might unexpectedly quit an application, or an application might take too long to respond to a command.

In some of these situations, you can figure out what went wrong by checking the error code that is returned or the line of code at which the application stopped. In other situations, you might have to retry a command a given number of times.

For example, the following code runs several command strings that are specified in code elsewhere. If an error occurs, such as a time-out error, the command is retried up to 10 times. If the command fails on the tenth try, the program displays an error message and stops.

```
Sub CallDDECommands(ExecuteStrings())
    On Error GoTo RetryCommand
    For Each Cmd In ExecuteStrings
        DDEExecute(ChannelNumber, Cmd)
    Next
    Exit Sub

RetryCommand:
    If Retries < 10 Then
        Retries = Retries + 1
        DoEvents                    ' Yield to other applications.
        Resume                      ' Try again.
    Else
        MsgBox Error(Err)           ' Display error message.
    End If
End Sub
```

When you are responding to an error by retrying the command, as in this example, it is a good idea to include a **DoEvents** statement, so that the operating system can let other applications run. Otherwise, your system might appear frozen while the command is retried.

For more information about handling errors, see Chapter 9, "Handling Errors and Error Values."

Sending Keystrokes

With the Microsoft Windows operating system, the easiest way to communicate with applications that do not support linking, embedding, OLE Automation, or DDE is by sending keystrokes.

Note You can send keystrokes to any running Windows-based application. However, you cannot send keystrokes to an application running in an MS-DOS session with Windows. You also cannot send keystrokes to applications running with the Macintosh operating system.

When you send keystrokes to another application, the result is the same as if the user were typing at the keyboard. Thus, you can make the other application perform any operation that the user can make it perform using keystrokes.

For example, this statement sends the characters "ABC" as if they were typed at the keyboard.

```
SendKeys String := "ABC"
```

When the **SendKeys** statement is used in this way, the keystrokes aren't sent until your procedure is *idle*—that is, until none of your Visual Basic code is running or until the **DoEvents** statement is called. However, you can force the keystrokes to be sent immediately, as shown in the following example.

```
SendKeys String := "ABC", wait := True
```

Passing **True** for the optional second argument on the call to **SendKeys** causes Visual Basic to yield, allowing the keystrokes to be processed, before continuing with the rest of your code.

The keystrokes go to the active application. If the active application is not the one to which you want to send keystrokes, then you need to switch to the other application.

Switching to Other Applications

Keystrokes that you send can go only to the active application. If you do not switch to another application, your procedure sends keystrokes to itself. To switch to another application, use the **AppActivate** statement. For example, if the Windows-based Terminal application is running, the following statement switches to it.

```
AppActivate title := "Terminal"
```

If the application is not already running, you must start it using the **Shell** function before you can switch to it.

Specifying Special Characters

You may want to send keystrokes that you cannot simply type into a string, such as ENTER, F1, and TAB. To send these keystrokes, you enclose the name of the key within brace characters ({}). For example, to send the keystrokes for TAB, F1, and ENTER, use the following statement.

```
SendKeys String := "{TAB}{F1}{enter}"
```

The names of the keys are not case-sensitive. For a list of these special keys, search the online Visual Basic Reference for *SendKeys statement.*

Note You cannot send keystrokes, such as ALT+CTRL+DEL and PRINT SCREEN, that generate interrupts rather than character codes.

Working from Other Applications

Up to this point, this chapter has discussed working with other applications from Microsoft Excel. This section explains how you work with Microsoft Excel from other applications. Many of these applications support the same mechanisms that Microsoft Excel uses.

The following table lists the ways that you can work with Microsoft Excel from some other Microsoft applications.

Application	OLE Automation	Linking and embedding	DDE	Sending keystrokes
Microsoft Access version 1.0	No	Yes	Yes	Yes
Microsoft FoxPro version 2.5	No	No	No	No
Microsoft Mail version 3.2	No	Yes	No	No
Microsoft PowerPoint® version 3.0	No	Yes	No	No
Microsoft Project version 4.0	Yes	Yes	Yes	Yes
Microsoft Publisher version 2.0	No	Yes	No	No

Application	OLE Automation	Linking and embedding	DDE	Sending keystrokes
Microsoft Visual Basic version 3.0	Yes	Yes	Yes	Yes
Microsoft Word version 6.0	No	Yes	Yes	Yes
Microsoft Works version 2.0	No	No	No	No

OLE Automation

When working in other applications that include Visual Basic for Applications, you can create and use objects as described in this book. For example, Visual Basic procedures written in Microsoft Project can use Microsoft Excel objects with the same syntax described in the Microsoft Excel documentation.

When working in other macro languages that support OLE Automation, you can create and use Microsoft Excel objects without special considerations.

Microsoft Excel adds the following class names to your system registration file. These names can be used to access objects through programming tools, such as Microsoft Visual Basic programming system for Windows, version 3.0.

Class name	Provides access to
Excel.Application	An Application object
Excel.Sheet	A Worksheet object
Excel.Chart	A Chart object

For more information about using class names with the **CreateObject** and **GetObject** functions, see the documentation for Microsoft Visual Basic programming system for Windows, version 3.0.

Linking and Embedding

Microsoft Excel provides the following OLE verbs for objects that are linked or embedded in other applications.

OLE verb	Purpose
"Activate"	Activates an embedded object for in-place editing
"Open"	Opens a linked object's source application for editing

DDE

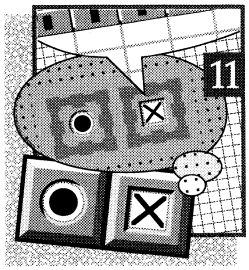
Microsoft Excel provides DDE commands for each of the Microsoft Excel 4.0 macro language functions. For example, the following code can be used in WordBasic to open a Microsoft Excel workbook. For more information about the available commands, see the online Macro Functions Help.

```
ExcelChannel = DDEInitiate("Excel", "System")
Cmd$ = "[OPEN(" + Chr$(34) + "C:\EXCEL\MYWORK.XLS" + Chr$(34) + ")]"
DDEExecute(ExcelChannel, Cmd$)
```

The next chapter explains how to use controls and dialog boxes as a first step in creating a custom application.

CHAPTER 11

Controls and Dialog Boxes



One important aspect of creating a custom application in Microsoft Excel is managing the way the user interacts with the application. You know about placing buttons and other graphic objects on worksheets and assigning macros to them. This chapter shows you some more sophisticated ways to exchange information between your application and its users through dialog boxes you create and support with Visual Basic procedures.

First, you will learn how to use simple predefined dialog boxes to enhance your application. Subsequent sections show you how to insert a new type of sheet—the dialog sheet—in a workbook and place objects such as buttons, edit boxes, and list boxes within it to create a custom dialog box that acts like the built-in dialog boxes supported by Microsoft Excel. You can also use these objects—called *controls*—to create your own data-entry forms and wizards.

You can use Visual Basic code to set and change the properties of controls and collect groups of controls together on a dialog sheet to create custom dialog boxes. You can also develop procedures that present a custom dialog box to a user and respond to the information entered in it. Finally, you can associate procedures with actions taken in a dialog box; for example, you can use a procedure to validate data entered in an edit box when the user presses the ENTER key.

Contents

Enhancing Your Application with Predefined Dialog Boxes

The easiest way to add a dialog box to your application is to use a predefined dialog box. A *predefined dialog box* is a kind of custom dialog box that you can quickly and easily design and display, but you have limited control over how it looks or what type of information you can get from or display to the user.

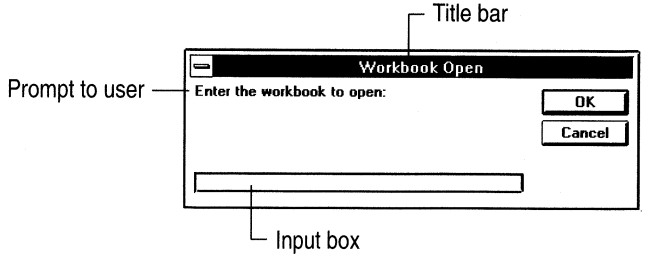
The following table lists the methods for adding predefined dialog boxes to your Visual Basic application.

Use this	To do this
InputBox function	Display a prompt in a dialog box and return the text typed by the user.
InputBox method	Display a prompt in a dialog box and return the information entered by the user. This method is similar to the **InputBox** function but provides additional functionality, such as requiring input to be of a specific data type.
MsgBox function	Display a message in a dialog box and return a value indicating the command button chosen by the user. The return value can be ignored.

Getting Information from the User

You use the **InputBox** function or the **InputBox** method to ask the user for information. When you use the **InputBox** function or method, you must work with the existing components of the dialog box. You can change only the text in the title bar, the prompt displayed to the user, and the position of the dialog box on the screen. To create a more elaborate dialog box, you must use a dialog sheet. For more information about creating a custom dialog box, see "Placing Controls on a Worksheet or Dialog Sheet" later in this chapter.

The **InputBox** function displays a dialog box that asks the user to enter a string or **Variant** variable. For example, the following dialog box contains an input box in which the user types the name of a workbook to open.



The following code shows how you can display the input box.

```
Sub OpenMyWorkbook()
    MyBook = InputBox("Enter the workbook to open:", "Workbook Open")
    Workbooks.Open filename:=MyBook
End Sub
```

For more information about the **InputBox** function, including the complete syntax, search the online Visual Basic Reference for *InputBox*.

Microsoft Excel also provides an **InputBox** method of the Application object that includes more features than the **InputBox** function. Specifically, you can use it to specify the type of information that the user can enter. For more information about the **InputBox** method, including the complete syntax, search the online Visual Basic Reference for *InputBox method*.

The following code uses the **InputBox** method and asks the user for the range to search and for the search value. The value must be a number.

```
Sub CountValues()
    Count = 0
    Set RangeToSearch = Application.InputBox( _
        Prompt:="Enter the range to search", _
        Type:=8)
    SearchValue = Application.InputBox( _
        Prompt:="Enter the search value", _
        Type:=1)
    For Each SpecificCell In RangeToSearch
        If SpecificCell.Value = SearchValue Then
            Count = Count + 1
        End If
    Next SpecificCell
    MsgBox CStr(Count)
End Sub
```

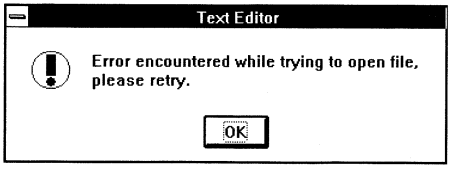
You can also display custom Help for the dialog box that the **InputBox** function or method displays. An example for the **InputBox** function is shown here.

```
' Use helpFile and Context. The help button is added
' automatically.
Value = InputBox(prompt:="Which item do you want to delete?", _
        helpFile:="DEMO.HLP", Context:=10)
```

For more information about displaying custom Help for a dialog box, search the online Visual Basic Reference for *InputBox* or see Appendix C, "Displaying Custom Help Topics."

Displaying Information with the MsgBox Function

Sometimes you might want to display brief messages such as errors or warnings. You use the **MsgBox** function to display these messages in a dialog box. After reading the message, the user chooses a button to close the dialog box.



The following code displays the preceding message box.

```
MsgBox "Error encountered while trying to open file, " & _
    "please retry.", vbExclamation, "Text Editor"
```

Note The **MsgBox** function returns a value indicating which button the user chose. For some dialog boxes, there might be no button choices, or you might not care what the return value is, so you have the option of ignoring it. In this case, **MsgBox** acts like a statement, not a function, and accordingly you do not enclose its arguments in parentheses. For more information about the **MsgBox** function, search the online Visual Basic Reference for *MsgBox*.

You can use the **MsgBox** function to create enhanced dialog boxes with different icons and buttons. You do this by adding built-in constants and using the sum as the *buttons* argument to the **MsgBox** function. You can also determine which button a user chose in the dialog box based on the return value of the **MsgBox** function.

The following example uses the **MsgBox** function to display a critical message in a dialog box containing a Yes and a No button. (A critical message, in this case, refers to a message that includes a Stop icon in its dialog box.) The No button is the default response. The variable `DialogStyle` contains the sum of the built-in constants that define the features of the dialog box (the *buttons* argument). The **MsgBox** function returns a value based on the button chosen by the user.

```
Sub CreateSampleBox()
    ' Put together a sample message box with all the proper components.
    Msg = "This is a sample of a critical-error message."
    Msg = Msg & " Do you want to continue?"
    'Describe dialog.
    DialogStyle = vbYesNo + vbCritical + vbDefaultButton2
    Title = "MsgBox Demonstration"
    Response = MsgBox(Msg, DialogStyle, Title)   ' Get user response.
    If Response = vbYes Then                     ' Evaluate response and
        Msg = "You chose Yes."                   ' act appropriately
```

```
    Else
        Msg = "You chose No or pressed Enter."
    End If
    MsgBox Msg                                    ' Display action taken.
End Sub
```

Note The constants vbYesNo, vbCritical, and so on, are defined in Visual Basic. For a list of these constants, search the online Visual Basic Reference for *MsgBox*.

Notice that the buttons and the critical message icon are created by adding the constants and storing the sum in the `DialogStyle` variable. Because `DialogStyle` is used as an argument to **MsgBox** in the line `Response = MsgBox(Msg, DialogStyle, Title)`, **MsgBox** displays the appropriate style of dialog box.

You can also include a Help button and display a custom Help topic for any dialog box displayed by **MsgBox**. For more information about custom Help topics, see Appendix C, "Displaying Custom Help Topics."

Placing Controls on a Worksheet or Dialog Sheet

Earlier in this book, we discussed placing a button or other object (such as a rectangle or picture) on a worksheet and assigning a macro that runs whenever the user clicks the object. For more information on placing buttons and other graphic objects on worksheets, see Chapter 13, "Creating Graphic Objects on Worksheets and Charts," in the *Microsoft Excel User's Guide*.

Similarly, the first step in adding a control to a custom application is to place it on a worksheet or dialog sheet. You can place controls on worksheets, chart sheets, and in custom dialog boxes, but not in a Visual Basic module.

One advantage of placing controls on a worksheet is that you can put them close to the data upon which the control is intended to act. For example, you could place a series of check boxes on a worksheet that simulates a 1040 form and support the sheet with a series of procedures that accesses tax tables, income worksheets, and so on. Another advantage of placing controls on a worksheet is that you can create a document that looks like a dialog box, but contains cells that recalculate automatically.

By using a dialog box to group controls together, on the other hand, you can reduce the degree of complexity that confronts the user working with a system of worksheets and procedures that you've developed. You can keep the user interface for your application uncluttered so the user isn't overwhelmed with options that are used infrequently. You can isolate the process of entering data from that of

transferring the data to a worksheet, in order to minimize errors on the worksheet. You can also create dialog boxes that apply across many worksheets or charts, or to the Microsoft Excel environment as a whole.

Using Custom Controls in Your Application

After you've placed a control, you set its initial properties, such as whether a check box appears checked or whether a button should be resized when an underlying cell on a worksheet is resized.

Next, you can assign a procedure to the event associated with a control. Events occur when the user selects a button, check box, or option button, or when the user edits the text in an edit box.

You can also link a control directly to a cell on a worksheet, without using procedures. This way, you can simplify the user interface for a worksheet, so the user can set options with the mouse rather than by typing data in a cell. For example, you can group near the top of a worksheet a series of check boxes that represents options in a feasibility study and then link each check box to a cell used elsewhere in the worksheet. If a user selects a check box, the value linked to the worksheet cell becomes **True** and the worksheet is recalculated. For more information about linking controls to cells, see "Linking Controls to Worksheets" later in this chapter.

Finally, you can manage the way controls work together by displaying or removing the dialog box containing them, by ensuring that the data entered in the dialog box is correct, and by making this data available to worksheets and other procedures in the Microsoft Excel environment. For more information about displaying a dialog box, see "Displaying a Custom Dialog Box" later in this chapter.

Creating a Dialog Sheet

To create a custom dialog box, you must first insert a new dialog sheet.

▶ **To insert a new dialog sheet**

1. From the Insert menu, choose Macro, and then choose Dialog.

   Microsoft Excel inserts a new dialog sheet before the current sheet in the active workbook and places within it a dialog box frame containing an OK and a Cancel button.

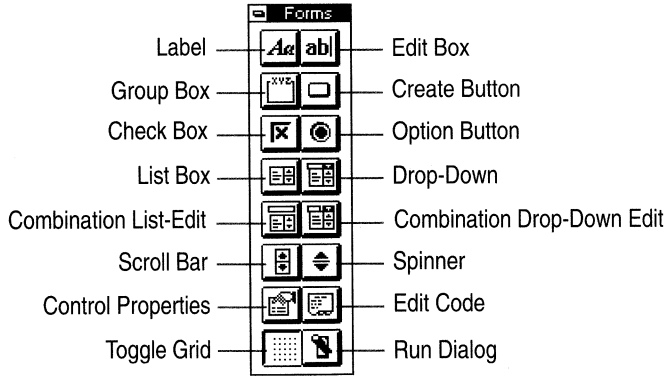
2. To change the name that will appear at the top of the dialog box, select the text at the top of the dialog box frame, and then type a new name.
3. To change the size of the dialog box, drag a handle until the dialog box frame is the size you want.

Note Even though you can change the location of a dialog box frame on its dialog sheet, Microsoft Excel displays the resulting dialog box in the same position as whatever dialog box you displayed last, regardless of whether that dialog box was built-in or custom. However, you can cause a custom dialog box to appear at a specific location by writing a procedure that positions the dialog box before displaying it, as discussed in "Displaying a Custom Dialog Box" later in this chapter.

You can view a dialog box as it will appear at run time by clicking the Run Dialog button on the Forms toolbar. When you do this, the dialog box becomes active. You can enter data in edit boxes, for example, and if you choose a button in the dialog box, its associated procedure runs.

Placing Controls

After you've created a new dialog sheet, you can use the Forms toolbar to place new controls in the dialog box frame; alternatively, you can place controls directly on a worksheet.



▶ **To place a control on a worksheet or dialog sheet**

1. On the Forms toolbar, click the button for the control you want to add.
2. On the worksheet or dialog sheet, drag until the control is the size and shape you want.

The following table describes the controls created by each corresponding button on the Forms toolbar.

Button name	Description of control
Label	Text you provide for the user, including names, instructions, and cautions.
Edit Box	A box in which a user can enter text, numbers, or cell references.
Group Box	A border containing a group of buttons or other controls.
Create Button	A command button such as the OK or Cancel button.
Check Box	A box that indicates whether an option is set, regardless of the state of other options in the dialog box.
Option Button	A button for selecting one of a group of mutually exclusive options. Place a series of option buttons in a Group Box to group them.
List Box	A list of text strings, one or more of which can be selected.
Drop-Down	A single uneditable text box and an arrow, paired with a drop-down list that appears when the user selects the arrow.
Combination List-Edit	A single editable text box combined with a list box.
Combination Drop-Down Edit	An empty edit box and an arrow, paired with a drop-down list that appears when the user selects the arrow.
Scroll Bar	A horizontal or vertical scroll bar for changing numeric values; to create a horizontal scroll bar, drag horizontally.
Spinner	A pair of buttons for incrementing or decrementing a displayed value.

The Forms toolbar contains four additional buttons for working with controls and dialog boxes.

Button name	Description
Control Properties	Used to view or change the properties belonging to the selected object; equivalent to choosing the Object command from the Format menu.
Edit Code	Used to create or edit code assigned to the selected object.
Toggle Grid	Used to align controls on a grid.
Run Dialog	Used to display the dialog box as it appears when run.

Note You can also use the Drawing toolbar to place standard drawing objects in a dialog box. For example, you can place a bitmap or other graphic object—such as a company logo—in a dialog box and then assign a procedure that runs when the user clicks the object. To display a hidden toolbar, use the Toolbars command on the View menu.

Selecting Controls

Typically, clicking a control on a worksheet or dialog box causes the same result you expect from a built-in dialog box control. For example, clicking a button on a worksheet causes the button to depress, and selecting a check box causes a check to appear in the box. If a control has an associated procedure, Microsoft Excel runs the procedure. If a control is linked to a cell on a worksheet, changing the contents or state of the control transfers the corresponding value to the worksheet.

When the control is selected, you can change its properties, move it around on the worksheet, or assign a procedure to one of the events it supports.

▶ **To select a control on a worksheet**

- Hold down CTRL in Microsoft Excel for Windows or COMMAND in Microsoft Excel for the Macintosh and click the control.

  You can also click the Drawing Selection button on the Drawing toolbar and then drag a rectangle around the control you want to select. If the toolbar is hidden, use the Toolbars command on the View menu to display it.

Note To select a group of controls on a dialog sheet, drag a rectangle that encloses the controls you want to select. You can also add controls to a group of controls that is already selected by holding down the SHIFT key and selecting the additional controls.

Sizing, Moving, and Deleting a Control

To change the dimensions of a control, select it and drag one of its handles. To change its position, drag one of its borders. A grid constrains the control's movement on a dialog sheet to specific increments, which vary with the type of display you're using; this behavior is called *snap-to-grid.*

▶ **To turn on or turn off snap-to-grid**

- Click the Toggle Grid button on the Forms toolbar.

You can temporarily override the snap-to-grid feature by holding down ALT in Microsoft Excel for Windows or COMMAND in Microsoft Excel for the Macintosh,

while dragging a handle, border, or control. Holding down ALT while dragging on a worksheet constrains movement to multiples of a cell width or cell height.

▶ **To delete a control**

1. Select the control.
2. Press DEL.

   You can also choose Clear and then choose All from the Edit menu.

Setting the Properties of a Control

When you create a control, Microsoft Excel assigns it a set of default properties like those found in the built-in Microsoft Excel dialog boxes. You can change some of these properties using the Object command on the Format menu. There are other properties that you can change only by running a Visual Basic procedure. For more information about modifying the properties of a control, see "Assigning Code to Controls and Dialog Boxes" later in this chapter.

▶ **To display or change a property of a control**

1. Select the control.
2. From the Format menu, choose Object.
3. Select a tab, and then review or change the property.
4. Choose the OK button.

Assigning Tab Order to Controls

There are several ways for a user to select a control in a dialog box (a process also called *setting the focus* to the control). One way is simply to click a control such as a button, edit box, or list box. Another is to press the ALT key in Microsoft Excel for Windows or the COMMAND key in Microsoft Excel for the Macintosh and then to click the access key for the control (discussed in the next section).

You can also establish a tab order for the controls in a dialog box. When the user presses the TAB key, controls become active in the sequence, or *tab order,* you've established. Establishing a logical tab order is particularly useful for custom data-entry forms, so that you can minimize the number of keystrokes it takes to get from one edit box to the next. You can set a tab order only for controls in a dialog box, not on a worksheet.

▶ **To change the tab order of controls**

1. Switch to the dialog sheet.
2. From the Tools menu, choose Tab Order.

3. Select the item whose place in the tab order you want to change.

   To select more than one control, hold down CTRL as you select items.
4. Choose the Move Up or Move Down button.
5. Choose the OK button.

Assigning an Access Key to a Control

An access key is the key you press with the ALT key in Microsoft Excel for Windows or the COMMAND key in Microsoft Excel for the Macintosh to select a control—equivalent to choosing a button. As with setting the tab order, defining access keys works only for controls in a dialog box. You can assign an access key to a label, command button, check box, option button, or group box. When you do this, a single character in the text associated with the control is underlined when the dialog box is displayed.

▶ **To set an access key for a control**

1. Select the control.
2. From the Format menu, choose Object.
3. Select the Control tab.
4. In the Accelerator box, type the key you want to be the access key.
5. Choose the OK button.

Note If you want to create the effect of an access key for a control placed on a worksheet, you can develop a procedure associated with the **OnKey** event that runs when the user presses a certain key combination. Creating an OnKey procedure is discussed in Chapter 13, "Creating Automatic Procedures and Add-in Applications."

Linking Controls to Worksheets

You can link the value of certain controls to a cell on a worksheet without developing a procedure and assigning it to the control; when the user changes the value in either location, the value is changed in both locations. For example, you can place on a worksheet a series of check boxes that sets global options for an entire workbook, with each check box linked to a cell on a different worksheet. If the user selects a check box, "TRUE" appears in the linked cell.

Worksheet cells can be linked to check boxes, option buttons, list boxes, scroll bars, and spinners. The link created is a property of the control, but not of the linked cell.

You can link the same cell to more than one control. However, you can link each control to only one cell, and that cell must reside on a sheet that is open in order for the link to work. To transfer a control's value to more than one cell, link the control to one cell, and then reference that cell in each cell where you want to use the value.

You can establish links for controls that you've placed in a dialog box. However, because links are always active, changing the state of the control always changes the value appearing in the linked cell, even if the user chooses the Cancel button to dismiss, or close, the dialog box. For this reason, it's usually a better idea to use a procedure to transfer information from a dialog box to the cells on a worksheet.

When you link a control to a worksheet cell, the worksheet is recalculated after the user enters a value in the cell or changes the state of the control. For example, if a check box is linked to a cell and a check appears in the check box, the value in the cell becomes **True**. When the value changes, those formulas referencing the cell are recalculated.

▶ **To link a control to a worksheet cell**

1. Select the control.
2. From the Format menu, choose Object.
3. Select the Control tab.
4. In the Cell Link box or Selection Link box, type the name or reference of the cell to which you want to establish the link.

   You can also click the cell to enter its reference in the box.
5. Enter any additional information that pertains to the control you want to link, as described in the following sections.
6. Choose the OK button.

Note You can also establish a link by selecting the control on the worksheet or dialog sheet and then typing in the formula bar the reference of the cell to which you want to establish the link.

You can link a control to a cell and assign a procedure to the same control. In this case, the link is updated before the procedure runs. Assigning a procedure to a control is described in "Assigning Code to Controls and Dialog Boxes" later in this chapter.

Check Boxes

The link established between a cell and a check box is *bidirectional*—changing the value of either the cell or check box changes the value that appears in the other location. The check box can appear checked, unchecked, or grayed. Similarly,

typing **TRUE** or a nonzero number in the linked cell causes the check box to appear checked; typing **FALSE** or **0** (zero) causes the check box to appear unchecked; and typing **#N/A** causes the check box to appear grayed.

Option Buttons

Like check boxes, option buttons can appear selected or not. Selecting one option button cancels the selection of the other option buttons in the group. Linking one option button in the group links every option button in the group. Similarly, entering a number in the linked cell selects the corresponding option button, and entering a number that doesn't correspond to an option button cancels the selection of every button in the group.

If you group more than one option button in a group box, the value returned to the linked cell is the ordinal number of the selected option button as it appears in the group box, counting from the back to the front. (You can use the Tab Order command on the Tools menu to change this order.)

If you don't surround the option buttons with a group box, all the option buttons on the worksheet or dialog box are treated as one group.

List Boxes

For list boxes, you specify a range of cells on a worksheet that contains the list of items to appear in the list box. You can also specify the reference of a single cell, in which appears the ordinal number of the item currently selected in the list box. Similarly, you can enter a number in the linked cell, and the corresponding item in the list becomes selected, although the list may not scroll to that item.

Formulas that depend on the linked cell are recalculated at the moment the selected item changes—this is useful for creating what-if models in which the user can watch results change while moving the selection up and down in the list box.

Scroll Bars

Scroll bars support a linked cell in which the position of the scroll box in the scroll bar is reported as a number. Using the Object command on the Format menu, you specify in the Format Object dialog box the starting and ending numbers and the amount the position number changes when the user clicks the arrows or the scroll bar. Changing this number in the linked cell changes the position of the scroll box.

Spinners

Spinners are similar to scroll bars, except that the user can click only the arrows; there is no bar. The value in the linked cell changes by the increment specified in the Format Object dialog box. Holding down the mouse button continuously changes the value in the linked cell; the worksheet is recalculated at every change.

Assigning Code to Controls and Dialog Boxes

After you place controls on a worksheet or in a dialog box and establish their initial properties, you need to develop a series of procedures that supports the dialog box.

The process of using a custom dialog box begins with showing it to the user, who then makes changes to the controls within it. Your procedures can access the state of the dialog box controls through their properties and then alter the way the application responds to the changes in the dialog box. You can validate the data entered, modify the behavior of controls or the dialog box itself while the dialog box is visible, or dismiss the dialog box and return the data entered in it to a worksheet.

Just as you assign a procedure to a button or graphic object on a worksheet, you can also assign a procedure to a control, using the **OnAction** property. In this case, the event is called an *event handler.* Whenever an action occurs on the control, Microsoft Excel runs the procedure assigned by the **OnAction** property. The following table describes the actions each control responds to.

Controls	Actions
Command buttons, group boxes, check boxes, option buttons	The control is activated; this event is used for controls that have no value or only a simple value, such as **True** or **False**.
Edit boxes, list boxes, drop-down boxes, combination list-edit boxes, combination drop-down edit boxes, scroll bars, spinners	The control is changed by the user. This event is used for controls that support a complex value.
Dialog boxes	The dialog box appears.

For example, you can use the **OnAction** property to assign a procedure to an edit box. The procedure can check the entered data whenever the user enters or changes a part number, so that only validated data can be returned to a worksheet. To validate all the data at once, you could assign a procedure to the OK button, and leave the dialog box visible until the user supplies the correct information.

▶ To associate an existing procedure with the event belonging to a control

1. Select the control.
2. From the Tools menu, choose Assign Macro.

   You can also click the control with the right mouse button to display the shortcut menu and then choose Assign Macro.
3. In the Macro Name/Reference box, type or select the name of the procedure you want to assign to the event.
4. Choose the OK button.

You can also create a new procedure and associate it with a control event.

▶ To create a new procedure and associate it with the event belonging to a control

1. Select the control.
2. Click the Edit Code button on the Forms toolbar.

   Microsoft Excel opens a Visual Basic module scrolled to display an empty procedure stub.
3. Between the **Sub** and **End Sub** statements, write the procedure you want to associate with the event.

For example, if you want to write an event handler for a Cancel button that is the fourth object in the dialog box frame, Microsoft Excel creates the procedure stub, and you might insert the following statement between the **Sub** and **End Sub** statements.

```
Sub Button4_Click()
    MsgBox "Are you sure you want to Cancel?", vbYesNo
End Sub
```

If there are no Visual Basic modules in the workbook containing the dialog sheet, Microsoft Excel inserts a new module and then switches to the programming stub.

▶ To edit a procedure previously associated with the event belonging to a control

1. Select the control.
2. Click the Edit Code button on the Forms toolbar.

   Microsoft Excel opens a Visual Basic module scrolled to display the associated procedure. If the associated code belongs to the Microsoft Excel 4.0 macro language, Microsoft Excel switches to the macro sheet containing the associated macro.
3. Edit the code.

Displaying a Custom Dialog Box

To display a dialog box, use the **Show** method in a procedure that has been assigned to a button, menu command, or event in the Microsoft Excel environment. For example, you could assign the following procedure to a button that resides on a worksheet.

```
Sub Button4_Click()
    DialogSheets("MyFileOpenDlg").Show
End Sub
```

Clicking the button invokes the **Show** method for the specified dialog box. When the dialog box appears, the user is free to interact with its controls — enter text into an edit box, select an option button, select items in a list box, and so on. If any of these controls has an associated **OnAction** event handler, the handler is executed.

If the user chooses the OK or Cancel button and the **DismissButton** property of the button is **True** and the **CancelButton** property is **False**, the following actions occur:

- The **OnAction** event handler for the button is executed, if one has been assigned.
- The contents of each edit box is validated, according to its Edit Validation property set on the Control tab of the Format Object dialog box.

  If the contents of an edit box are invalid, Microsoft Excel displays an alert message and switches the focus to the edit box so the user can supply a correct value. This process continues until every edit box contains valid data.
- The dialog box is removed, and execution continues with the first statement after the statement that used the **Show** method.

Usually you want to process the changes made in the dialog box only if the user chooses the OK button to dismiss the dialog box. In this case, the **Show** method returns **True**.

For example, suppose you've placed a button named Button 4 on a worksheet. To open a workbook, the user clicks this button to display a dialog box on a dialog sheet named MyFileOpenDlg. The procedure assigned to the button might look like the following example.

```
Sub Button4_Click()
    If DialogSheets("MyFileOpenDlg").Show Then
        FormatThisWorkbook
    End If
End Sub
```

When the user clicks Button 4, the procedure displays the MyFileOpenDlg dialog box and waits for the user to take an action. If the user chooses the OK button, **True** is returned to the **If...Then** statement and the FormatThisWorkbook procedure is executed. If the user chooses the Cancel button or presses ESC, **False** is returned and the procedure takes no action.

Note You can implement a dialog box in which no control has an associated event handler; if you do this, the procedure that displays the dialog box must process its results.

Creating an OnAction Event Handler

Whenever you display a custom dialog box, an **OnAction** event is generated for the dialog box. If you've created a procedure associated with the **OnAction** event, the procedure is executed before the dialog box appears. You can use this feature to initialize the controls in the dialog box to a set of default conditions, such as which control has the focus, or which option button in a group box is selected. Otherwise, the controls in the dialog box will remain in the state they were left the last time the dialog box appeared.

▶ **To associate a procedure with the OnAction event of a dialog box**

1. Select the dialog box frame by clicking its border.
2. From the Tools menu, choose Assign Macro.

   You can also click the dialog box with the right mouse button to display the shortcut menu and then choose Assign Macro.
3. In the Macro Name/Reference box, type or select the name of the procedure you want to assign to the event.
4. Choose the OK button.

For example, if a dialog box named InputDialog has a series of check boxes and an edit box named InputRange, you can create an **OnAction** handler that cancels the selection in the check boxes and sets the contents of the edit box to the contents of the first cell of Sheet1.

```
Sub DialogFrame1_Show()
    For Each c In DialogSheets("Dialog1").CheckBoxes
        c.Value = xlOff
    Next
    DialogSheets("Dialog1").EditBoxes("Edit Box 4").Text = _
        Worksheets("Sheet1").Cells(1, 1).Value
End Sub
```

Displaying a Built-in Dialog Box

Each built-in dialog box has an associated constant that names the dialog box. For a complete list of the reserved names for the built-in dialog boxes, search the online Visual Basic Reference for *Dialogs method.* Each built-in dialog box supports a list of arguments that you can use to set the initial state of its controls.

For example, if you are using Microsoft Excel for Windows, you can use the following code to display the built-in File Open dialog box.The default directory shown in the dialog box is XLDOCS.

```
Application.Dialogs(xlDialogOpen).Show("C:\XLDOCS")
```

If you are using Microsoft Excel for the Macintosh, you can use this code.

```
Application.Dialogs(xlDialogOpen).Show("MyDisk:XLDocs")
```

When the user chooses the OK button, Microsoft Excel carries out the actions specified by changes in the dialog box, as if the corresponding built-in menu item had been chosen. If the user chooses the Cancel button or presses ESC, execution continues at the next statement.

You can use the return value of the **Show** method to determine whether the user chose the OK or Cancel button. For example, the following routine executes the ProcessWorkbook procedure if the user has selected a workbook in the File Open dialog box.

```
Sub TestWindow()
    Application.Dialogs(xlDialogOpen).Show ("c:\")
    If ActiveWindow.Type = xlWorkbook Then myProcessWorkbookMacro
End Sub
```

Note You can duplicate a dialog box you've already created by copying the dialog sheet and then working with the copy. To reproduce the structure of a built-in dialog box in a custom version that supports a slightly different set of features, however, you must explicitly recreate the dialog box, control by control, and write a series of procedures that duplicates the features of the built-in dialog box. You can add or delete controls, or change the behavior of controls, through the procedures supporting the dialog box.

Getting Information from a Dialog Box

As discussed earlier in this chapter, you can use linking to transfer information from a control object to a cell on a worksheet. When a control is linked to a cell on a worksheet, the selected item in a list box or the value indicating the state of a check box is immediately updated on the worksheet containing the linked cell.

In the case of a dialog box, however, you can write procedures that take the information entered in the control, or a change in the state of the control, and make the new information available to the other procedures in your application.

For example, if a dialog box contains a single edit box, you can use the following code to transfer the data in the edit box to the active cell.

```
Sub EditBox4_Change()
    ThisText = DialogSheets("MyDialog").EditBoxes(1).Text
    Application.Sheets(2).Activate
    ActiveCell.Value = ThisText
End Sub
```

You can work with all the controls of a certain type by indexing their collection class. The collection class for option buttons, for example, is OptionButtons. If you want to read into an array the state of every option button in a dialog box, you can use the following code.

```
Sub ButtonChecking()
N = 0
For Each OptionBtn In DialogSheets(1).OptionButtons
    N = N + 1
    If OptionBtn.Value = xlOn Then
        Sheets("sheet1").Cells(N, 1) = "Checked"
    Else OptionBtn.Value = xlOff Then
        Sheets("sheet1").Cells(N, 1) = "Not checked"
    End If
Next
End Sub
```

Note A procedure can read and set the properties of a control at any time—before you display a dialog box, while it's visible, or after it has been closed.

Modifying Controls While a Dialog Box Is Visible

You change the initial properties associated with a control at design time using the Object command on the Format menu. You can also change any of the properties associated with a control while the procedure is running. For example, you might want to change the text that appears next to a check box if the user selects a certain option button, or make a series of options available when the user selects a check box.

Controls share many of the methods and properties that belong to drawing objects, such as dimension and location. Some properties, such as the **AutoSize** property (which determines whether a button can be resized when its underlying cell is resized), pertain only when the control has been placed on a worksheet. Other properties, such as the **Accelerator** property, pertain only when the control has been placed in a dialog box.

The complete list of properties for each control varies with the control. For more information about the properties that belong to each control, search the online Visual Basic Reference for the name of the control object.

Enabling a Control When a Condition Is Met

You can use the **Enabled** property of a control to prevent the user from making changes to an option unless a certain condition is met. A common use for this is to make a set of option buttons available when the user selects a check box, as shown in the following code, which is assigned to an **OnAction** handler for the check box.

```
Sub CheckBox6_Click()
    If DialogSheets(1).CheckBoxes(1).Value = xlOn Then
        DialogSheets(1).OptionButtons.Enabled = True
    Else DialogSheets(1).OptionButtons.Enabled = False
    End If
End Sub
```

Another example of a use for the **Enabled** property is to create an **OnAction** handler for an edit box that makes the OK button available when the user has entered a part number that conforms to a standard pattern.

Setting the Focus to a Control

You can set the focus to a control in a dialog box by setting the **Focus** property of the dialog box.

```
Sub DialogFrame1_Show()
    DialogSheets("MyFileOpenDialog").Focus = "Button 4"
End Sub
```

This gives Button 4 the focus when the dialog box is run.

Modifying Links Between a Control and a Worksheet

You can establish links between a control and a cell on a worksheet when the control is created, but you can also create, modify, and delete links between a control and a cell while a procedure is running. You use the **LinkedCell** property of the control to do this. The **LinkedCell** property belongs to the linked control and not the linked cell. The **ListFillRange** property also specifies the input range for list boxes.

Changing Dialog Box Properties

You can change some of the properties of the dialog box itself while it is visible, such as its size and position. A common use for this is to support an Options button. You can place controls below what is usually the bottom border of the dialog box. When the user chooses the Options button, the procedure changes the vertical size of the dialog box to reveal more options that user can set.

```
'Sets the expanded size.
Sub Button8_Click()
    DialogSheets("MyFileOpenDialog").DialogFrame.Height = 500
End Sub
```

Hiding a Custom Dialog Box

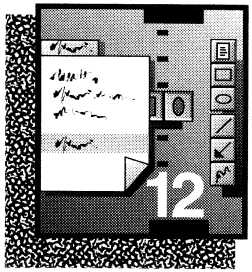
To remove a dialog box from the screen, you can either wait for the user to choose a button whose **DismissButton** property is **True**, such as the OK button, or you can create an event procedure for a control as shown in the following example.

```
Sub Button1_Click()
    N = MsgBox("Are you sure you want to close?", vbYesNo)
    If N = vbYes Then
        DialogSheets(1).Hide
    End If
End Sub
```

The next chapter explains how to customize menus and toolbars as another step in creating a custom application.

CHAPTER 12

Menus and Toolbars



Chapter 11, "Controls and Dialog Boxes," discusses adding controls to worksheets and dialog boxes and supporting them with Visual Basic procedures. This chapter presents two further enhancements you can add to the user interface of a Microsoft Excel application: custom menus and custom toolbars. While a group of complex options is best presented in a dialog box, an option that has one or two states—such as a check box that can appear with or without a check mark—is a good candidate for a custom menu command or toolbar button.

Microsoft Excel comes with a set of default menus presented in a system of menus, menu commands, and shortcut menus. Using the Menu Editor, you can change this menuing system to suit your own needs. You can add a menu to any menu bar and then add a menu command to that menu and assign it a procedure that runs when the user chooses the command. Your Visual Basic procedures can set and change the properties of a menu, such as whether a menu item has a check mark next to it or whether it is available to the user.

Toolbars provide a graphical counterpart to menu commands and work best for actions that are taken frequently. You can use the Button Image Editor to design toolbar buttons that picture an action to be taken. You can also create custom toolbars that work the same way the built-in toolbars do and assign procedures to toolbar buttons that run when the user clicks the button.

Contents

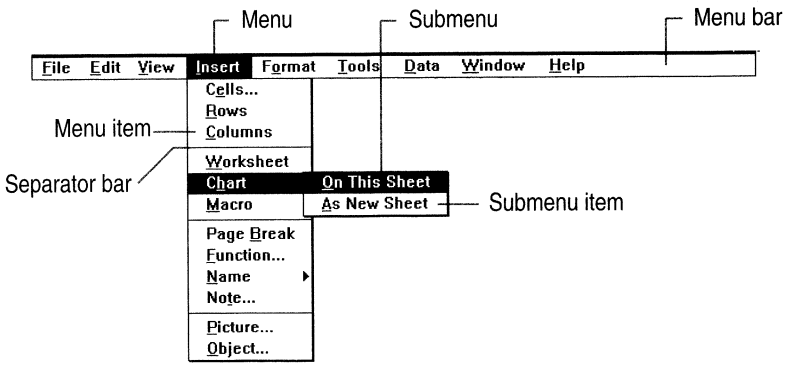
Modifying the Menuing System with the Menu Editor

The *menuing system* in Microsoft Excel is composed of the set of available menu bars, the menus on each menu bar, the menu items on each menu, and the submenus that belong to certain menu items. In this chapter, the term *menuing item* refers to any item in a menuing system: a menu bar, menu, menu item, submenu item, shortcut menu, or shortcut menu item.

You have already worked with the built-in menuing system. To insert a chart on a worksheet, for example, you find the Insert menu on the worksheet menu bar, choose the Chart menu item, and then choose the On This Sheet submenu item.

Note This chapter does not use the term command, because a command can be a menu item, a submenu item, or a shortcut menu item, and here it is important to maintain the distinctions between these objects.

Different elements of the menuing system appear in different contexts. When you edit a chart, for example, the menu bar changes to display a set of menus that applies to the charting environment. Similarly, you can assign procedures to the menu items, submenu items, and shortcut menu items you create and develop procedures that change the current menu bar to the one that's most appropriate for the sheet being displayed.



There are also shortcut menus that appear when the user presses the right mouse button (Windows) or holds down CTRL while pressing the mouse button (Macintosh) while the pointer is over an object. For example, if you click a chart object with the right mouse button, Microsoft Excel displays a shortcut menu, and you can use its menu items to act upon the chart or change its properties.

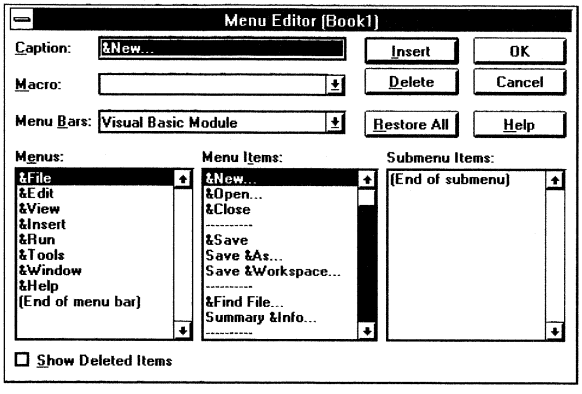
Creating Menuing Items

To modify the built-in menuing system, you can use either the Menu Editor or one or more Visual Basic procedures, as described later in this chapter. You can duplicate all of the functions of the Menu Editor dialog box through Visual Basic code, but unless you want to add or remove menuing items while your code is running, it's usually easier to use the Menu Editor to create a system of menus that is associated with the active workbook.

▶ To display the Menu Editor dialog box

- With a Visual Basic module active, choose Menu Editor from the Tools menu.

  You can also click the Menu Editor button on the Visual Basic toolbar with any worksheet active.



Creating a New Menu Bar

Usually you want only to add a single menu item to a built-in menu—a new menu item on the Tools menu, for example. You can also add an entirely new menu to a built-in menu bar and then add a series of menu items to that menu. These tasks are described in the following sections of this chapter.

If you want to create a custom application, however, you might need to create a new menu bar, complete with menus, menu items, and submenu items. You can add and remove menu items from any menu, whether built-in or custom, and you can add a custom menu to any menu bar, whether built-in or custom.

▶ To add a menu bar to a workbook

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the Menu Bars box, select the name of any menu bar.

   This step cancels the selection of items in the Menus, Menu Items, and Submenu Items boxes. If you omit this step, the text you type in the Caption box becomes the first menu item on the first menu that is selected in the Menus box.

3. Choose the Insert button.
4. In the Caption box, type the name of the new menu bar, or accept the default name if applicable.
5. Add any additional menu bars, menu items, or submenu items you want.
6. Choose the OK button.

The new menu bar is added to the Menu Bars list, below the final item (Shortcut Menus 3) in the list of built-in menu bars, which is shown in the following table. The custom menu bars you create are listed in order of creation. You can use up to 15 custom menu bars at one time.

Menu bar	Description
Worksheet	The menu bar that appears when a worksheet is active
Chart	The menu bar that appears when a chart is active
No Documents Open	The menu bar that appears when no workbooks are open
Visual Basic Module	The menu bar that appears when a Visual Basic module is active
Shortcut Menus 1	A group of shortcut menus that appears when the user clicks a toolbar, toolbar button, cell, column or row selection, workbook tab, window title bar, or the desktop with the right mouse button
Shortcut Menus 2	A group of shortcut menus that appears when the user clicks a drawing object, button, or text box with the right mouse button
Shortcut Menus 3	A group of shortcut menus that appears when the user clicks a chart series, text, plot area, axis, gridline, floor or arrow, legend, or the entire chart with the right mouse button

Note A shortcut menu is considered a type of menu, and each group of shortcut menus in the Menu Bars list is considered a type of menu bar. Also, to display a shortcut menu on the Macintosh, hold down CTRL and click the mouse button.

Because built-in menu bars are associated with specific types of sheets (for example, the charting menu bar appears when you switch to a chart), you might need to add a menu, menu item, or submenu item having the same name to more than one menu bar. For example, you might want to add to the File menu of each menu bar an Open Special menu item that presents a special dialog box for importing files that aren't supported by Microsoft Excel.

Creating a New Menu

You can add a menu to any built-in menu bar or to a menu bar you create. For example, you might want to add a special Accounting menu to each menu bar, so that employees in a company can run the corresponding macros from any sheet. You can also specify an access key for the menu; access keys appear underlined when the menu is displayed.

▶ **To add a menu to a menu bar**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the Menu Bars box, select the name of the menu bar to which you want to add the menu.
3. In the Menus box, select the name of the menu to the left of which you want to insert the new menu.

   –Or–

   To add a menu to the right of the rightmost menu, select End Of Menu Bar.
4. Choose the Insert button to create the new menu.
5. In the Caption box, type the name of the new menu.

   Type an ampersand (**&**) before the character you want to use as the access key for the menu.
6. Add any additional menu bars, menu items, or submenu items you want.
7. Choose the OK button.

Note You cannot add or delete a shortcut menu. However, if you select a shortcut menu group from the Menu Bars list, the Menus list displays the name of each type of object for which a shortcut menu is defined. By selecting from these, you can add items to the shortcut menu for an object.

Creating a New Menu Item or Shortcut Menu Item

You can add a menu item to a built-in menu, a shortcut menu, or a custom menu.

▶ **To add a menu item to a menu or shortcut menu**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the Menu Bars box, select the name of a menu bar or group of shortcut menus.
3. In the Menus box, select the name of the menu to which you want to add the menu item, or the object for which a shortcut menu is defined.
4. In the Menu Items box, select the name of the menu or shortcut menu item above which you want to insert the new menu item.

   –Or–

   To add a menu item to the bottom of the menu, select End Of Menu.
5. Choose the Insert button to create the new menu item.
6. In the Caption box, type the name of the new menu item you want to add.

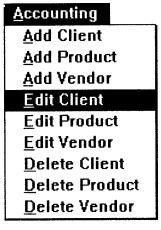
   Type an ampersand (**&**) before the character you want to use as the access key for the menu. To create a separator bar, type one hyphen (**-**).

7. To specify that a procedure run when the user chooses the menu or shortcut menu item, type or select the name of the procedure in the Macro box.
8. Add any additional menu bars, menu items, or submenu items you want.
9. Choose the OK button.

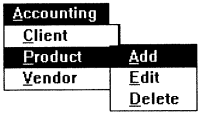
Creating a New Submenu

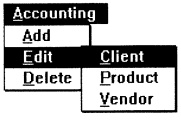
Submenus, also called cascading menus, appear when the user selects a menu item to which one or more submenu items are attached. With submenus, the user can browse through commands that might otherwise be available only through a series of custom dialog boxes.

You can use submenus to organize menus more effectively and add clarity to your application by reducing the amount of information presented to the user at any one time. For example, suppose you create a menu that presents a number of options, as shown in the following illustration.



Using submenus, you can present the same items in either of the following ways.





You can add submenu items to custom menu items, but not to built-in menu items. To add a submenu to a built-in menu item, you must first delete the built-in item, then replace it with a custom menu item that accomplishes the same task, and then associate the submenu item with the custom menu item.

▶ **To add a submenu item to a menu item**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the Menu Bars box, select the name of a menu bar.
3. In the Menus box, select the name of the menu to which you want to add the menu item.

4. In the Menu Items box, select the name of the menu item to which you want to add the submenu item.
5. In the Submenu Items box, select the name of the submenu item above which you want to insert the new submenu item.

   –Or–

   To insert a submenu item at the bottom of a group of submenu items, select End Of Submenu.
6. Choose the Insert button to create the new submenu item.
7. In the Caption box, type the name of the new submenu item.

   Type an ampersand (**&**) before the character you want to use as the access key for the menu.

   To create a separator bar, type one hyphen (**-**).
8. To specify that a procedure run when the user chooses the submenu item, type or select the name of the procedure in the Macro box.
9. Add any additional menu bars, menu items, or submenu items you want.
10. Choose the OK button.

Deleting, Restoring, and Renaming Menuing Items

You can remove items from any of the built-in menu bars, and you can delete custom menu bars, but you cannot delete a built-in menu bar. You can, however, delete all the menus on a built-in menu bar and restore them later if needed.

For example, you might want to replace a built-in Chart menu item with one that presents a custom chart wizard containing a series of special charting formats for your data. Or you might want to remove certain menu items so that inexperienced users of your application don't adversely affect its operation, or to make the user interface for your custom application easier to understand.

▶ **To delete a menuing item**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the appropriate box, select the name of the menu bar, menu, menu item, submenu item, or shortcut menu item you want to delete.
3. Choose the Delete button.
4. Choose the OK button.

You can restore built-in menu bars, menus, or menu items that you have deleted. However, custom menu bars, menus, or menu items that have been deleted cannot be restored.

▶ **To restore a deleted menuing item**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. Select the Show Deleted Items check box.

   Deleted menu items appear dimmed.
3. In the appropriate box, select the deleted item you want to restore.
4. Choose the Undelete button.
5. Choose the OK button.

Alternatively, you can restore all the built-in menu bars, menus, menu items, and submenu items by choosing the Restore All button. Doing this also deletes every custom menu bar, menu, menu item, and submenu item from the active workbook.

You can also rename custom menuing items.

▶ **To rename a menuing item**

1. With a Visual Basic module active, choose Menu Editor from the Tools menu.
2. In the appropriate box, select the item you want to rename.
3. In the Caption box, type the new item name.
4. Choose the OK button.

You cannot rename built-in items.

Where the Menuing System Is Stored

Built-in menuing items belong to the Microsoft Excel application. However, any of the changes you make to the menuing system—including added or deleted menuing items—are stored as a *menu editing list* in the active workbook. Each open workbook can have its own menu editing list, and the effects of that workbook's list on the menuing system combine with the menu editing lists stored in other open workbooks.

For example, suppose you have only one workbook open; you edit the menuing system in the workbook and develop procedures that support the menus, worksheets, and other objects in your application. If you then save and close the workbook, the menuing system reverts to the built-in state. When you open the workbook again, the menuing system is revised according to the menu editing list that was stored when the workbook was last saved.

Now consider the built-in File menu, in which the first three menu items are New, Open, and Close. Suppose you open a workbook called MYBOOK1.XLS and add an Open Special menu item to the File menu, just above the Close item. Then you close MYBOOK1.XLS and open a second workbook called MYBOOK2.XLS, to which you add an Open Database menu item just above the Close item on the File

menu. With MYBOOK2.XLS still open, if you then reopen MYBOOK1.XLS, the File menu contains all the menu items: New, Open, Open Database, Open Special (appearing here because its workbook was reopened second), and Close.

However, if you bring up the Menu Editor with MYBOOK1.XLS active and select the worksheet menu bar and the File menu, you see only the New, Open, Open Special, and Close menu items. If you bring up the Menu Editor with MYBOOK2.XLS active, you see New, Open, Open Database, and Close. This happens because in such a case the Menu Editor displays the effects of only the active workbook's menu editing list.

Managing Menus with Visual Basic

As mentioned earlier in this chapter, you can duplicate all of the functions of the Menu Editor with Visual Basic procedures. For most purposes, using the Menu Editor provides a simpler environment for managing menus. However, there are some tasks the Menu Editor cannot perform.

For example, you might need to display a menu item on a menu but make it unavailable to the user by disabling it (also called *dimming* the menu item or making it gray). Or, for conditions that can take either of two states, you might want to show that a certain condition is in effect by placing a check mark next to the corresponding menu item; for example, a check mark appears next to the Formula Bar menu item on the View menu when the formula bar is visible.

Finally, you might want to rename a menu item in response to current conditions. For example, choosing the Freeze Panes menu item on the Windows menu causes it to be renamed Unfreeze Panes. All of these tasks can be performed only through one or more Visual Basic procedures.

Working with Menu Bars

Visual Basic supports a variety of methods and properties for managing menu bars. The MenuBars collection contains all the menu bars available in the Microsoft Excel environment. You can use the **Count** property to find the number of menu bars available (but not the number of shortcut menu groups listed as menu bars in the Menu Editor dialog box).

You use the **Item** method to return a reference to a single menu bar, given the menu bar's index or name. You can use the following constants to specify the corresponding built-in menu bars.

Constant	Menu bar associated with
xlWorksheet	Worksheets and macro sheets
xlChart	Chart sheets

Constant	Menu bar associated with
xlNoDocuments	No documents open
xlInfo	Info window
xlModule	Visual Basic module

Although groups of shortcut menus are listed in the Menu Bars box of the Menu Editor dialog box, they cannot be accessed the same way. To return a reference to a shortcut menu, you use the **ShortcutMenus** method, as described later in this chapter.

Adding a New Menu Bar

You use the **Add** method to create a new menu bar, optionally specifying the name of the menu bar you want to add.

```
'Add a menu bar with the specified name
MenuBars.Add "MyWorksheetMenubar"
```

You can also use the **Index** property to find the index of a menu bar that you've added to the MenuBars collection, and the **Caption** property to set or find the name of a menu bar, given its index number. Because you can specify the name of a menu bar in the **Add** method, and because the index of a menu bar can be changed unexpectedly by adding or deleting new menu bars, you shouldn't need to use a menu bar's index.

You use the **Menus** method to get a collection of all the menus in the specified menu bar. For example,

```
MenuBars(xlWorksheet).Menus
```

contains the File menu, the Edit menu, and so on.

You can determine whether a menu bar belongs to the Microsoft Excel application or to a workbook by using the **BuiltIn** property, which returns **True** if the menu bar is built-in. For example, the following statement determines whether the active menu bar is built-in.

```
IsBuiltIn = ActiveMenuBar.BuiltIn
```

Displaying a Menu Bar

After you've created a menu bar, you use the **Activate** method to display it.

```
MenuBars("MyWorksheetMenubar").Activate
```

Microsoft Excel limits which of the built-in menu bars can be displayed, depending on the type of sheet that is active. For example, you cannot display the charting menu bar when a worksheet is active.

Removing a Menu Bar

You use the **Delete** method to delete a menu bar, given the name or index number of the menu bar.

```
MenuBars("MyWorksheetMenubar").Delete
```

As mentioned earlier in this chapter, you can delete only custom menu bars.

Restoring a Menu Bar

You use the **Reset** method to restore to its original configuration a built-in menu bar and all of the menuing items it contains.

```
MenuBars(xlWorksheet).Reset
```

Use the **Reset** method to restore a menu bar only if you're sure you no longer need any custom menu, menu item, or submenu item on that menu bar. Otherwise, your procedures must reconstruct the menu bar item by item, so that possible interactions with the menu editing lists belonging to other workbooks are minimized.

Working with Menus

The Menus collection contains the menus for a specified menu bar. Each menu bar maintains a **Menus** method that returns a collection of all the menus associated with it. For example:

```
Set ChartMenus = MenuBars(xlChart).Menus
```

returns a reference to a collection that contains the File, Edit, and View menus, among others.

Similarly, a Menu object defines a single menu on a menu bar, a shortcut menu item, or a submenu item. You can reference a menu by its name or by its position on a menu bar. Because the position of a menu on a menu bar can be changed by other procedures, by the Menu Editor, or by the effects of the menu editing list of a workbook other than the one containing the procedure, however, it's best to refer to menus by name rather than by position. For example, the following two expressions are equivalent only if the built-in Edit menu on the chart menu bar has not been modified.

```
Set SecondItem = MenuBars(xlChart).Menus("Edit")
Set SecondItem = MenuBars(xlChart).Menus(2)
```

Adding a Menu to a Menu Bar

You use the **Add** method to add a menu to a specified menu bar. For example, the following code adds a new MyWorkMenu to the left of the Help menu on the menu bar associated with a chart sheet.

```
Set NewMenu = MenuBars(xlChart).Menus.Add ( _
    caption := "&MyWorkMenu" , _
    before := "Help")
```

Use an ampersand (&) to specify the access key for the menu. After the menu has been added, you don't need to use the ampersand to reference the menu. To add the menu to the rightmost position, omit the *before* argument.

The **Add** method returns a reference to the new menu.

If you are using Microsoft Excel for the Macintosh, the **Add** method causes a trappable error if there isn't enough room for the name of the menu.

Deleting a Menu

You use the **Delete** method to delete a built-in or custom menu. For example, the following code deletes the Edit menu on every menu bar.

```
Sub RemoveEdit
    For Each mb In Menubars
        If mb.Index <> xlNoDocuments And mb.Index <> xlInfo Then
            mb.Menus("Edit").Delete
        End If
    Next mb
End Sub
```

Restoring a Built-in Menu

To restore a built-in menu item, you use the *restore* argument of the **Add** method. For example, you can restore the Edit menus deleted by the preceding example with the following code.

```
Sub RestoreEdit
    For Each mb In MenuBars
        If mb.BuiltIn = True Then
            If mb.Index <> xlNoDocuments and mb.Index <> xlInfo Then
                mb.Menus.Add Caption:="&Edit", Before:=2, Restore:=True
            End If
        End If
    Next mb
End Sub
```

In order to restore a custom menu, you must recreate it from scratch using the **Add** method, as described previously in this chapter.

Working with Menu Items and Submenu Items

The MenuItems collection contains the menu items on a specified menu, including separator bars and submenu items. The list of most-recently used files on the File menu counts as one item, and the list of open windows on the Window menu also counts as one item.

Similarly, a MenuItem object defines a single menu item. For example, the following statement sets the variable `FileNewMenuItem` to the New menu item on the File menu of the chart menu bar.

```
Set FileNewMenuItem = MenuBars(xlChart).Menus("File").MenuItems _
    ("New")
```

Adding a Menu Item to a Menu

You use the **Add** method to add a new menu item to the specified menu; through its arguments you can specify the menu's name, the name of the procedure to run when the user chooses the menu item, and its position on the menu. For example, you can add an Open Database menu item to the File menu with the following statement.

```
Set MenuItemAdded = MenuBars("MyMenubar").Menus("File").MenuItems _
    .Add (caption := "Open &Database" , _
        onAction := "OpenDatabaseProc" , _
        before := "Close")              'Appears before Close command.
```

To place the menu item at the end of the specified menu, omit the *before* argument.

To create a separator bar, you might use this statement.

```
'Hyphen creates the separator.
MenuBars("MyMenubar").Menus("File").MenuItems.Add _
    caption := "-" , _
    before := "Close"                   'Appears before Close command.
```

Deleting a Menu Item

You use the **Delete** method to remove a menu item from the specified menu. For example, the following code deletes the Open menu item from every File menu.

```
On Error Resume Next
For Each ThisMenu In MenuBars
    ThisMenu.Menus("File").MenuItems("Open...").Delete
Next
```

Enabling a Menu Item

You use the **Enabled** property to make a custom menu item available or unavailable to the user. If a menu item is not enabled (that is, if it's dimmed), the **Enabled** property returns **False**. For example, you can disable a custom MyOpen menu item on the File menu of the worksheet menu bar, without deleting it, using the following code.

```
MenuBars(xlWorksheet).Menus("File").MenuItems("MyOpen...") _
    .Enabled = False
```

Placing a Check Mark next to a Menu Item

You use the **Checked** property to display or remove a check mark next to a menu item, or to determine whether a menu item is checked: The **Checked** property returns **True** if the menu item is checked. For example, the following procedure is assigned to a menu item called Database added below the Toolbars menu item on the View menu belonging to the built-in worksheet menu bar. The procedure adds or removes a check mark by the Database menu item before switching between special views on a worksheet.

```
Sub DatabaseView()
    With MenuBars(xlWorksheet).Menus("View").MenuItems("Database")
        .Checked = Not .Checked
        If .Checked Then
            'Switch to database view
            .
            .
            .
        Else
            'Switch to worksheet view
            .
            .
            .
        End If
    End With
End Sub
```

Renaming a Menu Item

You can use the **Caption** property of a menu item to change the item's name in response to changing conditions in your Visual Basic code.

```
MenuBars("MyMenubar").Menus("File").MenuItems("Open Database") _
    .Caption = "Close &Database"
```

When you rename a menu item this way, make sure that the other procedures in your application reference the menu item by its new name, which is "Close Database" in the preceding example.

You can also use variables to refer to a menu item. An advantage of this technique is that variables continue to work even if the item's caption changes. For example, the following line of code sets a variable to the Open Database menu item.

```
Set MyMenu = MenuBars("My Menubar").Menus _
    ("File").MenuItems("Open Database")
```

You can change the caption later with the following code.

```
MyMenu.Caption = "Close & Database"
```

Restoring a Menu Item

You can restore a built-in menu item that has been deleted by using the **Add** method's *restore* argument. For example, the following code restores a deleted Open menu item on the File menu of the worksheet menu bar.

```
MenuBars(xlWorksheet).Menus("File").MenuItems.Add _
    caption := "Open" , _
    restore := True
```

If you omit the *restore* argument in this example, the statement adds a new Open menu item below the Exit menu item on the File menu.

Adding a Submenu Item

You use the **AddMenu** method to add a new submenu to a specified menu item, or to restore a deleted built-in submenu. As with the **Add** method, when you apply **AddMenu** to menu items you must specify the name of the submenu item using the *caption* argument. You can also specify the position of the item with the *before* argument, and you can restore a built-in submenu with the *restore* argument.

Working with Shortcut Menus

Unlike regular menus for worksheets and charts, shortcut menus are not contained on a menu bar. To access a shortcut menu, use the **ShortcutMenus** method. You specify a built-in shortcut menu by using its corresponding constant. (To find the constant for a built-in shortcut menu, search the online Visual Basic Reference for *ShortcutMenus*.)

For example, with the following code you can reference the shortcut menu that appears when the user clicks a chart series with the right mouse button (Windows) or while holding down CTRL and clicking a chart series (Macintosh).

```
ShortcutMenus(xlChartSeries)
```

There is no way to create a new shortcut menu; you can only modify a built-in shortcut menu through its MenuItems collection. For example, you can add a custom Format Special shortcut menu item at the end of the worksheet cell shortcut menu with the following code.

```
ShortcutMenus(xlWorksheetCell).MenuItems.Add _
    caption := "Format &Special"
```

Managing Toolbars and Toolbar Buttons with Visual Basic

A *toolbar* is the collection of controls that appears when you choose the Toolbars command from the View menu, select the check box next to the name of a toolbar, and then choose the OK button. Toolbars contain toolbar buttons. A toolbar button has two components—the button itself, which is the control the user clicks to initiate an action, and its button image, which is the graphic that appears on the toolbar button.

Many of the tasks you can perform with toolbars are described in Chapter 34, "Customizing Your Workspace," in the *Microsoft Excel User's Guide.* That chapter describes using the Customize option in the Toolbars dialog box to create a new toolbar or to add new or built-in toolbar buttons to a toolbar; using the Button Image Editor to customize the image associated with a particular toolbar button; and using the Assign Macro dialog box to assign a procedure that runs when the user clicks the toolbar button.

This chapter covers many of the same topics but discusses using Visual Basic code instead of the Toolbars dialog box or the Button Image Editor. As with the Menu Editor, it's usually best to use the Toolbars dialog box and the Button Image Editor to work with toolbars. However, constructing toolbars with Visual Basic procedures gives you more control if you need it.

You can write code to position toolbars on-screen, determine whether a toolbar button is available (enabled) or appears pressed, and associate custom toolbars with certain documents. You can also use the Attach Toolbars command on the Tools menu when a Visual Basic module is active to store a custom toolbar in a workbook.

Working with Toolbars

The Toolbars collection contains all the built-in and custom toolbars that are defined in the Microsoft Excel environment. You can use the Toolbars **Count** property to find the number of defined toolbars available. Similarly, a Toolbar object defines a single built-in or custom toolbar.

You can use the **Item** method to return a reference to a specific toolbar, given the toolbar's index or name. You shouldn't use a numeric index with the **Item** method, because the indexes associated with toolbars can change, and because every toolbar has a name.

You can use the **Name** property to set or return the name of a toolbar. If the toolbar is one of the built-in Microsoft Excel toolbars, the **Name** property is read-only. You can use the **BuiltIn** property to determine whether the toolbar is a built-in or custom toolbar.

Creating a New Toolbar

You use the **Add** method to create a new toolbar. You should always supply a name; if you don't, a name is supplied for you, in the form "Toolbar *n*," where *n* is a number.

For example, the following code creates a toolbar called MyAppToolbar but does not display it.

```
Toolbars.Add "MyAppToolbar"
```

Changing the Appearance of a Toolbar

Toolbars support several properties that you can use to resize them; to dock them at the top, bottom, left, or right edge of the workspace; or to position them elsewhere on-screen (if they are *floating* toolbars).

You use the **Width** property to set or return the width of the toolbar, in points. When read by a procedure, this property returns the actual width of the toolbar. You can also use the **Width** property to change the size of a toolbar; in this case, both the height and width change when the toolbar snaps to the specified width. When a procedure sets the **Width** property, the toolbar conforms as closely as possible to the specified width while still displaying all of its toolbar buttons.

Similarly, you can use a toolbar's **Height** property to return the height of the toolbar, in points.

The **Position** property sets or returns the position of the toolbar: whether it is docked on one of the four edges of the workspace (xlTop, xlLeft, xlRight, or xlBottom) or floating (xlFloating). A toolbar containing a drop-down list cannot be docked on the left or right side of the workspace.

If the toolbar is docked, you can use the **Top** property to set or return the distance, in points, from the top edge of the toolbar to the top edge of the toolbar's docking area. If the toolbar is floating (that is, its **Position** property is xlFloating), the **Top** property sets or returns the number of points from the top edge of the Microsoft Excel workspace to the top edge of the toolbar.

Similarly, if the toolbar is docked, you can use the **Left** property to set or read the distance, in points, from the left edge of the toolbar to the left edge of the toolbar's docking area. If the toolbar is floating (that is, its **Position** property is xlFloating), the **Left** property sets or returns the number of points from the left edge of the Microsoft Excel workspace to the left edge of the toolbar.

Displaying a Toolbar

After you've established the initial position of a toolbar, you can display it by setting its **Visible** property to **True**. This corresponds to selecting the check box next to the name of the toolbar in the Toolbars dialog box and then choosing the OK button. You can also determine whether a toolbar is displayed by reading the value of the **Visible** property.

For example, the following procedure, assigned to the MyAppToolbar menu item on the View menu, positions and displays the toolbar.

```
Sub ViewMyAppToolbar()
    With MenuBars(xlWorksheet).Menus("View").MenuItems("View MyToolbar")
        .Checked = Not .Checked
        If .Checked Then
            'Display the toolbar
            Toolbars("MyAppToolbar").Visible = True
        Else
            'Hide the toolbar
            Toolbars("MyAppToolbar").Visible = False
        End If
    End With
End Sub
```

When a toolbar is visible, the user can click any of the toolbar buttons on it to run the button's assigned procedure.

Deleting a Toolbar

You can use the **Delete** method to delete a custom toolbar from the Microsoft Excel workspace. If the specified toolbar is built-in, however, nothing happens, because you cannot delete a built-in toolbar. For example, the following code deletes the MyAppTools toolbar.

```
Toolbars("MyAppTools").Delete
```

Deleting a toolbar applies only to toolbars that belong to the Microsoft Excel workspace. Deleting a toolbar that belongs to a workbook must be done manually, through the Attach Toolbars dialog box, discussed later in this chapter.

Restoring a Built-in Toolbar to Its Default State

If the user or a procedure has modified one of the default toolbars, you can return the toolbar to its default state by using the **Reset** method. Using this method corresponds to choosing the Toolbars command from the View menu, selecting the check box next to the name of the customized built-in toolbar in the Toolbars dialog box, and then choosing the Reset button.

For example, you can reset all the toolbars to their default state and simultaneously delete all the custom toolbars using the following code.

```
For Each ThisToolbar In Toolbars
    If ThisToolbar.BuiltIn Then
        ThisToolbar.Reset
    Else
        ThisToolbar.Delete
    End If
Next
```

Working with Toolbar Buttons

The ToolbarButtons collection contains all the toolbar buttons on a specified toolbar. The **Count** property returns the number of buttons on a toolbar, including each gap (a separator between two buttons). The **Item** method returns a single button from a toolbar. Similarly, a ToolbarButton object defines a single button on a toolbar.

You can use the **BuiltIn** property to determine whether a given toolbar button is built-in; the property returns **True** if the button is built-in. After your procedure has determined that a toolbar button is built-in, you can use its **ID** property to find the identification (ID) number of the button. Similarly, the **Name** property returns, as text, the name of a built-in toolbar button.

For a list of the built-in toolbar buttons and their ID numbers, see Appendix D, "Toolbar Buttons in Microsoft Excel."

Adding a Toolbar Button to a Toolbar

You use the **Add** method to add a new button to an existing built-in or custom toolbar. Arguments to this method specify the following for built-in buttons:

- Name or ID number
- Position on the toolbar
- Name of a procedure to run when the user clicks the button
- Whether the button initially appears pressed or dimmed

For example, the following code adds two buttons, separated by a gap, to a toolbar called MyAppTools.

```
Set btns = Toolbars("MyAppTools").ToolbarButtons
' Add a button with the clock face
btns.Add _
    button:=213, _
    before:=1, _
    onAction:="Module1.MyScheduler", _
    enabled:=True, _
    pushed:=False
' Add a button with the trashcan face
btns.Add _
    button:=225, _
    before:=2, _
    onAction:="Module1.MyEmptyTrash", _
    enabled:=True, _
    pushed:=False
' Add a space between the buttons
btns.Add button:=0, before:=2
```

If you don't specify a position, the toolbar button is added to the end of the toolbar. If you don't specify a procedure to run when the button is clicked, and it is a built-in toolbar button, the default action occurs.

Copying and Moving Toolbar Buttons

You can use the **Copy** method to copy a button between toolbars or within a toolbar. The **Copy** method takes as an argument the destination of the toolbar button, expressed as a Toolbar object and a button position number. The buttons on the destination toolbar are shifted to the right (or down) to make room for the copied button. For example, the following code copies the toolbar button at the tenth position on the Standard toolbar to the fourth position on a toolbar called MyAppTools.

```
Toolbars(xlStandard).ToolbarButtons(10) _
    .Copy Toolbars("MyAppTools"), 4
```

To copy a button to the end of the toolbar, add one to the value returned from the **Count** property.

If you want to move a toolbar button rather than copy it, you can use the **Move** method, which takes toolbar and position destination arguments as does the **Copy** method.

For example, the following code moves the same toolbar button as in the previous example but removes the original copy.

```
Toolbars(xlStandard).ToolbarButtons(10) _
    .Move Toolbars("MyAppTools"), 4
```

Deleting a Toolbar Button from a Toolbar

To remove a button from a given toolbar, use the **Delete** method. For example, the following code deletes the third toolbar button from the Standard toolbar.

```
Toolbars("Standard").ToolbarButtons(3).Delete
```

Changing the Appearance of a Toolbar Button

After you've created and placed a button on a toolbar, you can alter its appearance in several ways: You can make the button available to the user; you can make it appear as if pressed; or you can work with its button image.

You use the **Enabled** property to set or return whether a toolbar button is enabled (can be clicked) or disabled (is dimmed and beeps when clicked). For example, the following code disables the third button on the Standard toolbar.

```
Toolbars("Standard").ToolbarButtons(3).Enabled = False
```

You can use the **Pushed** Property to set or return whether a button appears as if pressed; the **Pushed** property returns **True** if the toolbar button appears as if pressed. A good use for this, analogous to placing a check mark next to a menu item, is to indicate that a condition is currently in one of two possible states. For example, the following procedure, assigned to a new Database View toolbar button, changes the appearance of the toolbar button before switching between special views on the worksheet.

```
Sub DatabaseView()
    With Toolbars("MyAppToolbar").ToolbarButtons(3)
        .Pushed = Not .Pushed
        If .Pushed Then
            'Switch to database view
            .
            .
            .
        Else
            'Switch to worksheet view
            .
            .
            .
        End If
    End With
End Sub
```

You can make it possible for the user to edit a button image by using the **Edit** method for the button, which brings up the Button Image Editor. For example, the following code makes it possible for the user to edit the graphic on the third toolbar button.

```
Toolbars("Standard").ToolbarButtons(3).Edit
```

You use the **CopyFace** method to transfer a bitmap image associated with a toolbar button to the Clipboard. After you've placed a button image on the Clipboard, either by copying it from another application or by copying the button image associated with an existing toolbar button, you can use the **PasteFace** method to paste a button image from the Clipboard onto a specified button.

You can use the **BuiltInFace** property to determine whether the button is using its built-in button image or a custom face that was pasted over the built-in face. If a procedure sets the value of this property to **True**, the pasted face is removed, returning the toolbar button to its default button image.

Finally, you can use the **LargeButtons** property of the Application object to specify larger-format toolbar buttons. Setting large buttons is useful on computers that have large monitors or monitors that display a large number of pixels per inch.

Attaching Toolbars to Workbooks

You can move a toolbar that you've created from the Microsoft Excel workspace to a workbook. Workbook-level toolbars make it easier to create a polished user interface for a custom application, such as an add-in, or to move custom toolbar buttons and their supporting procedures from one environment to another.

▶ **To move a toolbar to a workbook**

1. With a Visual Basic module active, choose Attach Toolbars from the Tools menu.
2. In the Custom Toolbars box, select the name of the toolbar you want to copy to the active workbook.
3. Choose the Copy button.

   The name of the toolbar appears in the Toolbars In Workbook box.
4. Choose the OK button.

You can also delete the original workspace-level toolbar by choosing Toolbars from the View menu, selecting the checkbox next to the name of the toolbar you want to delete, and then choosing the Delete button. If you don't delete the workspace copy of the toolbar, you can change it without affecting the copy stored in the workbook. You can, of course, copy the edited workspace toolbar to the workbook again, replacing the copy.

After you copy a toolbar to a workbook, the toolbar becomes available only after the user has opened that workbook. A workbook toolbar retains not only the name and contents of the toolbar, but also the assignment of code to buttons; the location, size, and shape of the toolbar; its position on-screen (docked or not); and whether it is in a visible or hidden state.

You can also delete a workbook toolbar.

▶ **To delete a workbook toolbar**

1. With a Visual Basic module active, choose Attach Toolbars from the Tools menu.
2. In the Toolbars In Workbook box, select the name of the toolbar you want to delete.
3. Choose the Delete button.
4. Choose the OK button.

Where Toolbars Are Stored

When you exit Microsoft Excel, the toolbars in the workspace are saved in EXCEL5.XLB (in the \WINDOWS directory in Microsoft Excel for Windows) or in EXCEL TOOLBARS (5) (in the PREFERENCES folder in the SYSTEM folder in Microsoft Excel for the Macintosh). The toolbars saved in a workbook are stored in the workbook file.

When you open a workbook that contains one or more toolbars, Microsoft Excel first determines whether a workspace toolbar with that name already exists. If not, Microsoft Excel creates a new workspace toolbar and copies the workbook toolbar into it. This way, the user gets a fresh copy of the toolbar and can alter it by hiding it or copying toolbar buttons to or from the workspace-level copy of the toolbar. When the user exits Microsoft Excel, changes made to this copy are stored in the EXCEL5.XLB or EXCEL TOOLBARS (5) file.

There is no way to rename the toolbar, so when the workbook or add-in is opened again, the toolbar from the workbook is not copied again, because the workspace already contains a toolbar with that name. However, the procedures that support the toolbar buttons in the open workbook still run if the user clicks the corresponding toolbar button.

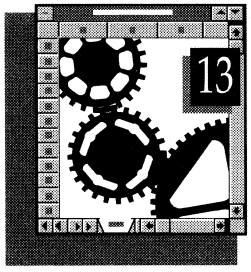
Because toolbars are stored and copied this way, developers of add-in applications can distribute workbooks containing unmodified copies of toolbars, so that users can move toolbar buttons to their own custom toolbars. A developer of an add-in workbook can design a toolbar using the Toolbar dialog box and the Button Image Editor and then attach the toolbar to the add-in workbook using the Attach Toolbars dialog box.

When the user opens the add-in workbook, the custom add-in toolbar appears. The user can then edit it and move toolbar buttons from it to personal toolbars, without affecting the copy stored in the add-in workbook. The user's changed toolbars are stored in the EXCEL5.XLB or EXCEL TOOLBARS (5) file when the user exits Microsoft Excel. When the user starts Microsoft Excel again, the edited toolbar appears; clicking one of the developer's toolbar buttons loads the add-in workbook containing the procedure attached to the toolbar button. To generate a fresh copy of the workbook toolbar, the user can delete the edited copy.

The next chapter concludes the discussion about creating custom applications by explaining events, automatic procedures, and add-in applications.

CHAPTER 13

Creating Automatic Procedures and Add-in Applications



This chapter discusses the last major mode of interaction between the user and Microsoft Excel, and between Microsoft Excel and other applications: events. An *event* is the occurrence of an action such as opening a workbook, switching to a sheet, using a particular key combination, or recalculating a worksheet. Some events are initiated by the user; others are initiated by Microsoft Excel or by other applications. By assigning procedures to events, you can enhance or alter the way users interact with your application.

There are three main classes of events, organized by the way you implement the procedures associated with the events:

- You can use the Assign Macro command on the Tools menu to associate a procedure with the action of clicking a button or other object, as discussed in the preceding chapters.
- You can assign procedures to run automatically when certain events occur by using a naming convention that begins with Auto_.
- You can create an OnEvent procedure (for example, using the **OnWindow** or **OnCalculate** property) that runs when its associated event occurs.

This chapter also shows you how to package Microsoft Excel worksheets, controls, menus, toolbars, and supporting Visual Basic modules in the form of an add-in application. You can use add-in applications to assemble and distribute custom features that—from the user's point of view—act as if they are built into Microsoft Excel itself.

Contents

Creating Automatic Procedures

An *automatic procedure* is stored in a workbook and runs automatically, such as when the workbook is opened or closed. You can place automatic procedures in any Visual Basic module in the workbook. Workbook-level automatic procedures are identified by their names as they appear in the module—Auto_Open or Auto_Close.

▶ **To define an automatic procedure**

1. Switch to the Visual Basic module where you want to store the automatic procedure.
2. Create a new procedure named Auto_Open or Auto_Close.
3. Write the code for the procedure.

To test an Auto_Open or Auto_Close procedure without having to explicitly open or close the workbook, you can choose Macro from the Tools menu, select the name of the procedure, and then choose the Run button.

You can use at most one Auto_Open and one Auto_Close procedure in each workbook. If more than one Auto_Open or Auto_Close procedure exists, none of them will run.

Auto_Open Procedures

An Auto_Open procedure runs when a workbook is opened. You can use an Auto_Open procedure to set up menu bars (if you've created menu bars with a procedure rather than with the Menu Editor), to customize the workbook for the operating system on which it is running, to display a custom startup screen, or to initiate links to other files or other applications.

For example, the following procedure automaticallyadds a custom menu item to the worksheet Tools menu.

```
Sub Auto_Open()
    MenuBars(xlWorksheet).Menus("Tools").MenuItems.Add _
        caption := "My Analysis", _
        onAction := ThisWorkbook.Name & "!Module2.MyAnalysisProc", _
        before := 1
End Sub
```

Auto_Close Procedures

An Auto_Close procedure runs just before the workbook containing the procedure closes. You can use this feature to restore menu bars, toolbars, and other elements of the user interface to their previous condition and to save and close the files that support an application.

You can also use an Auto_Close procedure to ensure an orderly disconnection from another application. For example, if your application uses Windows Terminal to download stock quotations from a remote information service, you probably want Terminal to log off from the service before closing the workbook, as shown in the following procedure.

```
Sub Auto_Close()
    MenuBars(xlWorksheet).Menus("Tools"). _
        MenuItems("My Analysis").Delete
    CloseLogFile        'Procedure: close transaction file.
    DisconnectService   'Procedure: disconnect info service.
End Sub
```

Using Defined-Name Automatic Procedures

Microsoft Excel supports automatic procedures that are associated with specific worksheets rather than the whole workbook. In most cases, it's better to use standard automatic procedures rather than defined-name procedures, because when using the latter you must explicitly define procedures for each worksheet that you want to support, and because these procedures are then not available for chart sheets and Visual Basic modules.

▶ **To create a defined-name automatic procedure**

1. Switch to the worksheet for which you want to define an automatic procedure.
2. From the Insert menu, choose Name, and then choose Define.
3. In the Name box, type a name that begins with Auto_Open, Auto_Close, Auto_Activate, or Auto_Deactivate. Precede the name with the sheet name following by an exclamation point—for example, **Sheet1!Auto_Activate_MyActivate**
4. In the Refers To box, type an equal sign and the name of the procedure you want to associate with the defined name, such as **=Book1!Module1.CheckData**.

   This procedure runs when the worksheet is activated or deactivated or when the workbook is opened or closed. You can use the same name you typed in the Name box, or a different name.
5. Choose the Add button.

Defined-name automatic procedures differ from standard automatic procedures in that the name defined on the worksheet needs only to begin with the name of the type of the procedure you want to run for the event. For example, you can define the names Auto_Open_Public, Auto_Open_StartupScreen, and Auto_Open_Files in the same worksheet, and, when the workbook containing the worksheet is opened, the procedures associated with the three names are run. If you want, you can hide a defined name to prevent accidental or deliberate tampering.

Standard automatic procedures run before defined-name procedures. For example, suppose you place an Auto_Open procedure in a module and in a certain worksheet define the name Auto_Open_ThisWorksheet as the text StartupWorksheet. When the user opens the workbook, the workbook-level Auto_Open procedure runs before the StartupWorksheet procedure. Also, you cannot set the Auto_Open procedures defined in each worksheet to run in any specific order.

Note You can prevent an automatic procedure from running by pressing the SHIFT key while opening, closing, or switching to or from a sheet or a workbook. For example, you can press SHIFT while opening an application that is under development to avoid running any of its Auto_Open procedures.

Opening a Workbook Automatically When Microsoft Excel Starts

If a workbook has been placed in the Microsoft Excel startup directory or folder, the workbook opens automatically when Microsoft Excel starts. The startup directory in Microsoft Excel for Windows is XLSTART; the startup folder in Microsoft Excel for the Macintosh is EXCEL STARTUP. For more information about using a startup directory or folder, see Chapter 35, "Controlling What Happens When You Start Microsoft Excel," in the *Microsoft Excel User's Guide.*

You can use the **StartupPath** property of the Application object to return the path of the startup directory or folder, the **AltStartupPath** property to set or return the path of the alternate startup directory or folder, and the **Path** property to read the path where the Microsoft Excel program is located.

For example, in Microsoft Excel for Windows, these properties take the form `C:\EXCEL`. In Microsoft Excel for the Macintosh, these properties take the form `HD80:MICROSOFT EXCEL`.

Use the Application object's **PathSeparator** property to return the appropriate separator for specifying the location of directories in Windows (\) and folders on the Macintosh (:).

Creating OnEvent Procedures

An *OnEvent procedure* (also called an event handler) is a procedure that runs when a specified event occurs. This type of event is an action generated by the Microsoft Excel environment but not necessarily by the user, such as the completion of a specified interval of time, the activation of a window, the recalculation of a worksheet, or the pressing of a specific key sequence.

You've already encountered one type of OnEvent procedure in Chapter 11, "Controls and Dialog Boxes." Assigning a procedure to the **OnAction** property of a control is discussed in that chapter.

Chapter 9, "Handling Errors and Error Values," and Chapter 10, "Controlling and Communicating with Other Applications," also discuss OnEvent procedures. Chapter 9 covers using an **On Error** statement to trap various types of errors encountered during the execution of a Visual Basic procedure, and Chapter 10 discusses using the **SetLinkOnData** method to receive data from applications outside Microsoft Excel.

The process for creating and using an OnEvent procedure is similar, regardless of the type of event.

▶ **To define an OnEvent procedure**

1. Write a procedure (the event handler itself) to be executed when the event occurs.
2. In another procedure, use a Visual Basic method or property (the **OnTime** method, for example) in a statement that associates the event with its event handler.

After the statement that associates the event with its handler is executed, the handler runs whenever the event occurs; this is called *trapping* the event. To turn off event trapping, you use the same Visual Basic method or property to disassociate the event and its handler.

The following sections show how event trapping is done with specific types of events.

OnTime Method

An OnTime event handler runs at a specified date and time, but only if Microsoft Excel is running and the workbook containing the OnTime procedure is loaded. You can use the **OnTime** method of the Application object to run an OnTime event handler at a specific time of day or after a specific period has passed. You can specify the time you want the handler to run and the name of the handler. For example, to accumulate and print a series of reports every day at noon, you might use the following code.

```
'Initialize trapping using OnTime method
Sub TrapTime()
    'Set OnTime arguments
    Application.OnTime _
        earliestTime := TimeValue("12:00:00"), _
        procedure := "DoReports"
End Sub

Sub DoReports()             'OnTime handler
    AssembleReports
    PrintReports
End Sub
```

The TrapTime procedure initializes trapping of the **OnTime** event; at the first noon after TrapTime runs, Microsoft Excel runs the DoReports handler. To run the procedure only once, set the *earliestTime* argument to a date and a time rather than using the time alone.

If another procedure is running when the event occurs, Microsoft Excel waits for an additional interval specified by the *latestTime* argument of the **OnTime** method. For example, the following version of the TrapTime procedure waits for half an hour for Microsoft Excel to become free and then stops the event.

```
Sub TrapTime()                    'Initialize OnTime trapping.
    Application.OnTime _
        earliestTime := TimeValue("12:00:00"), _
        procedure := "MyOnTimeHandler", _
        latestTime := TimeValue("12:30:00")
End Sub
```

If you do not supply the *latestTime* argument, the procedure runs when Microsoft Excel next becomes available. You can schedule more than one OnTime handler to run at different times; if you schedule more than one OnTime handler for the same time, the last-scheduled procedure runs first.

Tip The initialization of an **OnTime** event is not stored from session to session. To achieve the effect of its being stored, you can place in the startup directory or folder a workbook that contains both an Auto_Open procedure and the OnTime handler itself. The Auto_Open procedure must initialize OnTime trapping for a specific time. Then, whenever Microsoft Excel is started, trapping of that event is reinitialized.

To remove a pending **OnTime** event, you must specify the exact time of the event and the name of its handler and set the *schedule* argument to **False**. For example, if there are two **OnTime** events scheduled for noon, one handled by DoReports and a second handled by a procedure called CleanUp, you can stop the CleanUp handler request with the following code.

```
Application.OnTime _
    earliestTime := TimeValue("12:00:00"), _
    procedure := "CleanUp", _
    schedule := False
```

Note With the **OnTime** method, the user can work until an **OnTime** event occurs, unlike with the **Wait** method, which doesn't allow user interaction with Microsoft Excel. To suspend all activity except for printing and recalculation, use the **Wait** method as shown in the following statement.

```
'Wait 15 secs.
Application.Wait Now + TimeValue("00:00:15")
```

For more information about the **Wait** method, search the online Visual Basic Reference for *Wait method.*

OnSheetActivate and OnSheetDeactivate Properties

An OnSheetActivate handler runs whenever the user switches to a sheet in an open workbook. This event handler does not run when a procedure activates the sheet. You can set the **OnSheetActivate** property for the Application, Workbook, or any sheet object in order to run a procedure when the user switches to any sheet in the application, any sheet in a specific workbook, or a specific sheet in a workbook.

You can use **OnSheetActivate** to ensure that a toolbar that belongs to the workbook is displayed whenever the user switches to that workbook. For example, the following Auto_Open procedure in a workbook initiates trapping for the **OnSheetActivate** event for the workbook and runs a procedure that displays an Engineering Analysis toolbar whenever the user switches to the workbook.

```
Sub ShowAnalysisToolbar()
    ThisWorkbook.OnSheetActivate = "WBActivateHandler"
End Sub

Sub WBActivateHandler()
    If ActiveSheet.Name = "My_Data" Then
        Toolbars("EngAnalysisToolbar").Visible = True
    Else
        Toolbars("EngAnalysisToolbar").Visible = False
    End If
End Sub
```

Another use for an OnSheetActivate procedure is to display a custom menu bar when the user switches to a sheet of a certain type. For example, if the user switches to a chart sheet, the built-in chart menu bar is displayed automatically. If you create a custom chart menu bar, Microsoft Excel does not display it unless you create a procedure that runs whenever the user switches to a chart sheet.

Similarly, an OnSheetDeactivate procedure runs when the user switches away from a sheet or workbook. OnSheetDeactivate procedures are useful for resetting a customized user interface, for example returning menu bars, toolbars, and view settings to their original states.

Note The next workbook becomes active before the OnSheetDeactivate procedure in the previous workbook runs. Using the **ActiveSheet** or **ActiveWorkbook** property from the OnSheetDeactivate handler might therefore produce unwanted results.

OnWindow Property

An OnWindow event handler runs whenever the user switches to the specified window, or when the window belonging to the Microsoft Excel application is activated or opened by the user. Remember that you view a whole workbook through a window. (To assign a procedure to a single sheet in a workbook, use an Auto_Activate procedure.)

For example, the following code contains a procedure that traps each instance of the user switching to the second window called BikeDB.XLS and forces the window to appear at a specific location on-screen.

```
Sub TrapBicycle2()
    Windows("BikeDB.XLS:2").OnWindow = "PositionWindow"
End Sub

Sub PositionWindow()
    With Windows("BikeDB.XLS:2")
        .Left = 0
        .Top = 100
        .Width = 300
        .Height = 50
    End With
End Sub
```

To disassociate a window from its OnWindow handler, set the **OnWindow** property to "", as shown in the following example.

```
Windows("BikeDB.XLS:2").OnWindow = ""
```

To associate a procedure with the user's switching to any window in the Microsoft Excel application, use the **OnWindow** property of the Application object, as shown in the following example.

```
Application.OnWindow = "AllWindowHandler"
```

If a workbook, worksheet, or other sheet has an associated OnSheetActivate procedure, it runs after the procedure specified in the **OnWindow** property.

OnKey Method

An OnKey event handler runs when the user presses a specified key combination. To associate a procedure with a specified key combination, use the **OnKey** method of the Application object. For example, the following code runs the DoReports procedure when the user presses the F12 key.

```
Sub TrapKeys()                          'Trap key combinations.
    Application.OnKey _
        key := "{F12}" , _
        procedure := "DoReports"
End Sub

Sub DoReports()                         'F12 OnKey handler.
    AssembleReports
    PrintReports
End Sub
```

For the complete list of special codes for key combinations, search the online Visual Basic Reference for *OnKey Method.*

Note An OnKey handler does not run if its associated key combination is pressed while another procedure is running. So you cannot use an OnKey handler if you want to run a special procedure when the user presses ESC to stop the currently running procedure. (In Microsoft Excel for the Macintosh, the user can also press COMMAND+PERIOD to stop the procedure.)

OnCalculate Property

An OnCalculate event handler runs immediately after a worksheet is recalculated. You can use the **OnCalculate** property of either the Worksheet or Application object to associate a procedure with the recalculation of a single, specified worksheet or with the recalculation of any open worksheet.

For example, you might use an OnCalculate handler to update column widths when new data is recalculated, as shown in the following example.

```
Sub TrapCalculate()
    Application.OnCalculate = "FitColumns"
End Sub
```

```
Sub FitColumns()
    Columns("A:H").EntireColumn.AutoFit
End Sub
```

If you've assigned a procedure to the **OnCalculate** property of the Application object, the procedure runs for any recalculated worksheet that doesn't have its own OnCalculate handler. To disassociate recalculation with a worksheet's handler, set its **OnCalculate** property to "".

OnEntry Property

An OnEntry handler runs when the user enters data in a worksheet. You can use the **OnEntry** property of either the Worksheet or Application object to associate a procedure with the entry of data on any worksheet. The event occurs after the user enters data in a cell and presses ENTER or selects another cell. The event does not occur if the user chooses the Cut or Paste command from the Edit menu, or if another procedure changes the contents of a cell.

For example, the following code associates with the OnEntry event a procedure that validates the data entered in a cell, but only if the cell is in column B.

```
Sub TrapEntry()
    ActiveWorkbook.Worksheets("GeneCountDB").OnEntry = _
        "ValidateColB"
End Sub

Sub ValidateColB()
    With ActiveCell
        If .Column = 2 Then              'Test for second column.
            If IsNumeric(.Value) Then
                If .Value < 0 Or .Value > 255 Then
                    MsgBox "Entry must be between 0 and 255."
                    .Value = ""
                End If
            Else
                'Handle non-numeric entry
                MsgBox "Entry must be a number between 0 and 255."
                .Value = ""
            End If
        End If
    End With
End Sub
```

OnData Property

An OnData handler runs when data arrives from an application other than Microsoft Excel. You can use the **OnData** property of the Worksheet, Application, Link, or Subscriber object to associate a procedure with the arrival of data linked to one of these objects through dynamic data exchange (DDE) or object linking and embedding (OLE). (In Microsoft Excel for the Macintosh running under System 7, data can also arrive through the Publish and Subscribe feature.)

For example, the following code associates the arrival of stock price data in a worksheet with a procedure that validates each item of information.

```
Sub TrapData()                          'Set up OnData trapping
    Worksheets("StockAnalysis").OnData = "ValidateData"
End Sub

Sub ValidateData()                      'OnData handler
    For Each Cell in InputRange         'InputRange is a Public variable
        CheckData                       'Run my validation procedure
    Next
End Sub
```

For more information about using DDE and OLE, see Chapter 10, "Controlling and Communicating with Other Applications."

Note An OnData handler does not run if it's linked to another running copy of Microsoft Excel.

OnRepeat and OnUndo Methods

An OnRepeat event handler runs when the user chooses any of the various forms of the Repeat command from the Edit menu; for example, after pasting text into a cell, the user can choose the Repeat Paste command to repeat the operation. A good use for an OnRepeat event handler is after the user runs a macro that performs a series of complicated operations on a worksheet.

You can use the **OnRepeat** method of the Application object to customize the form the Repeat command takes and to associate a procedure with the Repeat menu command.

For example, suppose that after pasting a date into a cell, you want to increment the date by a day in each successive paste taken from the cell above the active cell. To do this, you might use code such as that shown in the following procedure.

```
Sub TrapRepeat()
    Application.OnRepeat _
        text:= "Paste Next Day", _
        procedure := "PasteIncrement"
End Sub

Sub PasteIncrement()                        'OnRepeat handler.
    With ActiveCell
        .Value = .Offset(-1, 0) + 1         'Add a day.
    End With
End Sub
```

Similarly, an OnUndo event handler runs when the user chooses the Undo command from the Edit menu. You use the **OnUndo** method of the Application object to customize the form the Undo command takes and to associate a procedure with the action of choosing the Undo command.

For example, in a procedure that inserts a row containing data for a new record, the last action might be to enter data in a cell; when this happens, the Undo command reads Undo Entry. But choosing this command removes the information entered in the cell, when you'd probably rather delete the entire row with a command named Undo New Record instead.

Creating an Add-in Application

You can use an add-in application to distribute copies of your work in an unreadable, unmodifiable form that others can use. In its simplest form, an add-in might contain a series of user-defined functions. A more complex add-in application might contain a custom dialog box, custom menus, a unique toolbar, and a series of **Function** and **Sub** procedures that knit the application together. If a user places an add-in application in the startup directory or folder, the add-in's features act as if they are built-in features. Your user-defined functions, for example, appear in the Function Wizard dialog box (displayed when you choose Function from the Insert menu).

Converting a Workbook to an Add-in Application

When you save a workbook in the add-in format, Microsoft Excel converts its code into a compressed form that cannot be read by anyone, and cannot be converted back into a workbook. This process is called *compiling*. Compiling speeds execution of the resulting code and protects your work from being read or modified by anyone else.

Add-in applications support many of the characteristics of workbooks, but not all. The procedures in your add-in can use data in its worksheets and can copy internal charts to paste into an open workbook, but the user cannot edit or view these sheets.

Important Because add-in applications cannot be edited, it is very important to keep a copy of your original workbook so that you can update its procedures or add new procedures. Therefore, be sure to choose a different filename when saving a copy of a workbook as an add-in application. (In Microsoft Excel for Windows, if you use the suggested .XLA extension for the add-in, the filename of the add-in is different from that of the workbook. In Microsoft Excel for the Macintosh, the suggested filename of the add-in is the same as the current filename, followed by "Add-In.")

Before you save a copy of a workbook in the add-in format (assuming you've already thoroughly tested and debugged your code), you should remove all debugging code, such as statements that use the **Stop** and **Debug.Print** methods.

▶ **To create an add-in application**

1. With a Visual Basic module, Microsoft Excel 4.0 macro sheet, or dialog sheet active in the workbook that you want to convert, choose Make Add-In from the Tools menu.
2. In the Directories box in Microsoft Excel for Windows or the Folders box in Microsoft Excel for the Macintosh, select the directory or folder in which you want to save the file.
3. In the File Name box, type the name you want for the compiled add-in file.
4. Choose the OK button.

Transporting an Add-in Application

When you move an add-in application to another computer, you must include all of the type or object libraries and supporting files and workbooks that are referenced from the add-in. Add-in applications are platform-independent because both the Windows-based and Macintosh versions of Microsoft Excel can interpret the compiled code. (However, type libraries are platform-specific.)

For more information about object libraries and developing code for international applications, see Appendix A, "Writing Code for International Use."

Using an Add-in Application

Add-in applications are *demand-loaded,* which means that certain elements of the add-in are loaded immediately, before the entire add-in is loaded. For example, if an add-in contains a menu editing list (a custom list that adds or deletes menu items), the effects of the menu-editing list on the current menuing system appear when the user starts Microsoft Excel. When the user actually chooses a menu item specific to the add-in, Microsoft Excel loads the entire add-in from the disk.

However, if an add-in contains an Auto_Open procedure, the complete add-in is loaded when Microsoft Excel starts.

When you install an add-in, its menu editing list and attached toolbars are read and stored in the EXCEL5.INI file in Microsoft Excel for Windows or in the EXCEL PREFERENCES (5) file in Microsoft Excel for the Macintosh. When you next run Microsoft Excel, all the menu items for the add-in application appear, even though the add-in isn't loaded. Choosing one of its menu items causes the add-in to load. When you remove the add-in application, its menu-editing list is deleted from EXCEL5.INI or EXCEL PREFERENCES (5).

When you open a workbook, its menu-editing list is used to revise the current menuing system before any of the Auto_Open procedures in the workbook are run. With this feature, you can write Auto_Open procedures that work with the menu commands that have been added.

Similarly, when you run Microsoft Excel after installing an add-in, key combinations that have been assigned to procedures are available to the user, the names of user-defined functions appear in the Function Wizard dialog box (Insert menu), and the names of macros appear in the Macro dialog box (Tools menu), even though the add-in application might not be loaded.

Controlling Add-in Applications with Visual Basic

The AddIns collection contains a list of all the add-in applications available to Microsoft Excel, regardless of whether they are installed. This list corresponds to the list of add-ins displayed in the Add-Ins dialog box (Tools menu). Similarly, an AddIn object consists of a single add-in application, installed or not installed.

To access other add-in applications, use the **Add** method of the AddIn object, which adds a new add-in application to those available to Microsoft Excel. Doing this corresponds to choosing the Browse button in the Add-Ins dialog box (Tools menu) and then selecting the name of a file. For example, the following code makes an add-in application called MYADDINS.XLA available to Microsoft Excel for Windows but does not install it.

```
AddIns.Add fileName := "C:\MYAPP\MYADDINS.XLA"
```

The **Installed** property of the AddIn object sets or returns **True** if the add-in is installed; this property corresponds to the check box opposite the name of the add-in application in the Add-Ins dialog box. Setting this property to **True** installs the add-in. When the add-in is installed, its Auto_Add functions are called. Setting this property to **False** removes the add-in. When the add-in is removed, its Auto_Remove functions are called.

```
AddIns("MYADDINS").Installed = True          'Install MYADDINS
AddIns.Add("C:\MYAPP\MYADDINS.XLA").Installed = True
```

Add-in applications share many of the properties common to workbooks in general, such as the **Author**, **Title**, **Subject**, **Keywords**, and **Comments** properties of the Workbook object.

The **Name** property of the AddIn object returns the filename of the add-in. The **FullName** property returns the name of the add-in, including its full path; this is equivalent to the **Path** property followed by the current path separator and the add-in's **Name** property. The **Path** property returns the path to the add-in application, without the final separator.

Putting It All Together

A Microsoft Excel application is a package of procedures, sheets, and user-interface elements that, taken together, accomplishes a single, overall purpose. Some applications might consist of nothing more than a series of **Function** procedures that you want to distribute as add-in applications; when the user adds them to the Microsoft Excel environment, the functions are indistinguishable from built-in functions. Other applications might consist of a single toolbar button supported by one or more procedures that perform a single task.

A full-blown vertical market application, however, such as an inventory-control system for a small business, might include all of the following:

- Specially formatted worksheets that contain supporting data and defined names and chart templates for displaying results
- Menuing items, such as custom menu bars, menus, and commands
- Toolbars and toolbar buttons
- Dialog boxes containing controls that are supported by a series of procedures
- Visual Basic modules containing the procedures that support your application

You can package the workbook containing these items, hide it, protect it, and then convert it into an add-in application.

Sample Application

The Microsoft Excel installation disks contain a workbook that demonstrates many of the principles discussed in this chapter. It manages a simple database that tracks inventory in a bookstore and consists of the following elements:

- A worksheet containing a database in which several fictitious records are stored
- A dialog box in which the user can retrieve and edit records from the database
- A toolbar and menu items
- A Visual Basic module containing the procedures that manage the application

The filename is BOOKST.XLS in Microsoft Excel for Windows or BOOKSTORE INVENTORY in Microsoft Excel for the Macintosh.

APPENDIX A

Writing Code for International Use

Visual Basic makes it easy for you to exchange Microsoft Excel applications (usually add-in applications) with users in other countries whose native language is not the one in which you wrote your code. In this appendix, an *application* is assumed to be a workbook or collection of workbooks, with its related documents, that performs a specific task and that runs within Microsoft Excel.

This appendix explains how to choose a language for writing your code and how to write and distribute applications for international use. You will also learn techniques to ensure that your code runs as you expect it to run, and more important, as your users expect it to run, no matter where you transport it.

Making Your Code Transportable with Object Libraries

You can write Visual Basic code in many languages. No matter which language you choose, your code can run on other-language versions of Microsoft Excel (subject to the limitation discussed in the following note). This is possible because Visual Basic uses object libraries when it compiles your code. An *object library* is a file that determines what words and symbols Visual Basic can understand—that is, the names of objects, properties, methods, Visual Basic statements, and so on that you type in a Visual Basic module. These words and symbols vary from language to language.

Important You cannot transport code written on a system that is set up for a Far Eastern double-byte character system (DBCS) language to a system that is not similarly set up, unless the code is written in English.

In addition to providing vocabulary information in object libraries, operating system settings provide information that Microsoft Excel uses to determine how specific countries work with dates and times and which symbols are used for currency values, the decimal separator, the thousands separator, and the list separator. The combined language and country information identifies a *locale*. For example, there

are English/Australia, English/Canada, French/France, and French/Canada locales. When you write international code, think in terms of locales, not just languages and not just specific countries.

With object libraries, you can write your code in a way that is comfortable for you, and users in other countries can run your code in a way that is comfortable for them. For example, if the period is the decimal separator in your locale, you want to use it when you're typing numbers in code. If another symbol, such as the comma, is the decimal separator in your user's locale, you want it to appear when your code displays numbers. And you want your code to interpret the comma correctly when a user enters a number into a cell or dialog box. Object libraries make this possible.

Choosing a Locale in Which to Write or Edit Visual Basic Code

If the appropriate object libraries are installed on your disk and registered with Visual Basic, you can choose the language in which you want to write your code. You can also specify a corresponding country that determines how you type dates and times, what symbols you use for the list separator and decimal separator, and so on. The language/country combination specifies the locale of your Visual Basic environment. (This locale might or might not be the same as the operating system locale that affects the output of your code when it is run.)

▶ **To choose a locale in which to write or edit Visual Basic code**

1. From the Tools menu, choose Options.
2. Select the Module General tab.
3. In the Language/Country box, select the language and country you want to use.
4. Choose the OK button.

If the locale setting you want is not shown in the Language/Country box, then you must install the appropriate object libraries, as explained in "Installing and Registering Object Libraries" later in this appendix.

For users who do not want to write code using one of the predefined locales, Visual Basic provides "(Custom)" language choices in step 3. In this case, Visual Basic interprets the date formats, list separators, and so on, in your code based your operating system locale settings. This is not something you usually want to do, because applications created with custom settings are not likely to run on other systems.

Under the International group on the Module General tab are Current Settings and Default Settings option buttons, along with a display area showing the list separator, the number format, and so on. Selecting the Current Settings button displays the Language/Country and associated settings for the active workbook. Selecting the Default Settings button displays the default settings for new workbooks you create. So, for example, if you usually work in the English/United States locale and you receive a French/France application, selecting Current Settings displays locale information about the application, but the default settings for the rest of your work remain English/United States.

Working in Multiple Locales

One locale setting is stored with each workbook, so different workbooks can have different settings, even if they are all open at the same time. With this feature, you can write and call procedures written in several languages.

Visual Basic does not automatically translate existing code from one language into another. You must edit the code using the same locale setting that was used when the code was originally written. Using multiple locales requires using multiple workbooks. (However, you can edit the code with any language version of Microsoft Excel; for example, you can edit Dutch code using Norwegian Microsoft Excel, but the code is always displayed in Dutch.)

Installing and Registering Object Libraries

If you cannot find the locale you want to use listed in the Language/Country box on the Module General tab, then you must install and register the corresponding object libraries.

Visual Basic in Microsoft Excel requires at least two files for each language in which you write or run code—the Microsoft Excel object library and the Visual Basic for Applications object library. Microsoft Excel always comes with the English versions of these libraries, as well as those for your language version of Microsoft Excel (for example, Italian Microsoft Excel includes the English and Italian object libraries, a total of four files).

The names of the object library files and how you acquire the files are discussed later in this appendix.

▶ **To install object libraries**

- For each language you want to work with, copy the two object library files to your hard disk.

The following locations are recommended for your object library files:

- If you're using Microsoft Excel for Windows, copy the object library files (*.OLB) to the SYSTEM subdirectory of the directory in which you installed Microsoft Windows. (A common path is C:\WINDOWS\SYSTEM.)
- If you're using Microsoft Excel for the Macintosh, copy the object library files to the MICROSOFT SHARED FILES subfolder of the SYSTEM FOLDER or of the EXTENSIONS folder.

If someone gives you an application written in another language and includes the object library files, those files are registered automatically when you try to run any of the procedures in the application. However, if you're writing code and using the object libraries for the first time, you must register them with Microsoft Excel yourself. *Registering* files means to make Microsoft Excel aware that they exist. You have to register an object library only once, unless you move or rename it.

▶ **To register a new language version of the object library files**

1. Switch to a Visual Basic module.
2. From the Tools menu, choose References.
3. Choose the Browse button.
4. In the Directories box (or Folders box on the Macintosh), click the directory or folder names until you locate the new Microsoft Excel object library file.

   The filenames are listed in "Names of the Object Library Files" later in this appendix.
5. In the File Name box, type or select the object library filename.
6. Choose the OK button.
7. Repeat steps 3 through 6 for the Visual Basic for Applications object library file.
8. Choose the OK button to close the References dialog box.

You can now choose the Options command from the Tools menu, select the Module General tab, and select the locale you want.

If the language you want is still not available, then a language has already been assigned to your workbook. You should open a new workbook and repeat the preceding steps. Remember, you can assign a language to workbook only once, as explained in "Working in Multiple Locales" earlier in this appendix.

Distributing Object Libraries

Unless your Visual Basic code is written in English, you should always include the appropriate language object libraries when you distribute your application, just in case the recipients don't already have the library files. If your code calls procedures

written in multiple languages, then you should include the object libraries for each of those languages. (Microsoft allows all registered owners of Microsoft Excel to distribute object libraries with their applications to other registered owners as necessary. The names of the object libraries are listed in the following section, "Names of the Object Library Files.") You should also include all workbooks and other object libraries that are referenced by the application you are distributing.

Object libraries are platform-specific, so if you plan to distribute your workbooks to users who run Microsoft Excel for Windows and Microsoft Excel for the Macintosh, then you must include the Windows and Macintosh versions of the object libraries.

Tip To determine the location of an object library on your disk, switch to a Visual Basic module, choose the References command from the Tools menu, select the library name under Available References, and note the Path listed under Group. This procedure is also useful for determining which object library files you should distribute with your application—the check boxes next to them will be selected.

Names of the Object Library Files

For Microsoft Excel for the Macintosh, the object library filenames are Excel English Lexicon and Lexique Français Excel, respectively. The object library filenames for other languages substitute the appropriate language name in place of "English."

For Microsoft Excel for Windows, some of the object library filenames are listed in the following table.

Language	Microsoft Excel for Windows object library filename	Visual Basic for Applications object library filename
English	XLEN50.OLB	VBAEN.OLB
French	XLFR50.OLB	VBAFR.OLB
German	XLDE50.OLB	VBADE.OLB
Spanish	XLES50.OLB	VBAES.OLB
Italian	XLIT50.OLB	VBAIT.OLB
Portuguese	XLPTG50.OLB (Portugal)	VBAPTG.OLB
Portuguese	XLPTB50.OLB (Brazil)	VBAPTB.OLB
Swedish	XLSV50.OLB	VBASV.OLB
Norwegian	XLNO50.OLB	VBANO.OLB
Finnish	XLFI50.OLB	VBAFI.OLB
Dutch	XLNL50.OLB	VBANL.OLB
Danish	XLDA50.OLB	VBADA.OLB

Many other object libraries might become available later.

Obtaining New Object Library Files

To obtain new Microsoft Excel object library files, contact one of the following sources:

- CompuServe. After you connect to CompuServe, go to the Microsoft Software Library (type **Go MSL**), search for the keyword *object* or *library,* and then download the files you need.
- Microsoft Excel users in other countries with whom you exchange applications.

For information about obtaining object library files for other desktop applications (such as Microsoft Project or Microsoft Word), see the documentation for those applications.

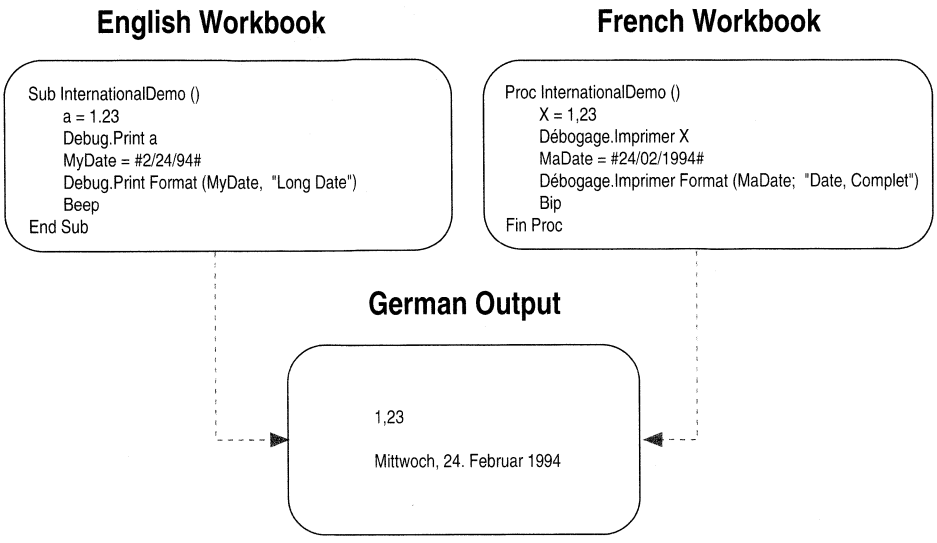
Note Microsoft Word version 6.0 does not include separate object library files. As a result, when you enter Visual Basic code in Microsoft Excel to control Microsoft Word, you must use the language that corresponds to the language version of Microsoft Word. For example, if you're trying to control French Microsoft Word, you must use French WordBasic statement and function names, regardless of any Visual Basic locale settings.

Running Your Code in Other Language Environments

The preceding sections explain how a locale setting determines how you, the programmer, should enter Visual Basic code. However, the way you enter items such as dates, formats, and so on, is not always the same as what the user should see, depending on his or her locale.

At run time, Visual Basic functions usually behave as they should for the user. Visual Basic usually uses the locale settings of the operating system (not the locale settings of your Visual Basic editing environment) to determine how to interpret numbers and dates entered by your users, such as information they enter into dialog boxes, values they enter in cells, and so on. Visual Basic also uses these settings to determine how to display to those users numbers and dates, such as the results of the **Format** function.

For example, the following two procedures are written in different languages, one in English and one in French, but they display the result in the same language when they're run on the same system. The language of the display is determined by the locale settings of the operating system. The following illustration assumes the user has a German system with German, English, and French object libraries installed.



For more examples of the differences between what you, the programmer, create and what the user in another locale sees, see "Guidelines for Writing Transportable Visual Basic Code" later in this appendix.

As a user, you might occasionally need to change your operating system locale settings to suit your needs. Or, as a programmer, you might want to change the settings to test applications that you plan to distribute to others. Changing operating system locale settings changes the output of locale-aware functions and statements.

▶ **To change the locale (country) settings in Microsoft Windows**

1. Open the Microsoft Windows Control Panel.
2. Double-click the International icon.
3. In the Country box, select the name of a country.

   The list separator and date, time, number, and currency formats automatically change as appropriate, although you can change each one yourself.
4. Choose the OK button.
5. Close the Control Panel.

In the International dialog box, you can also specify a language if you have your Microsoft Windows Setup disks with the appropriate language dynamic-link library (DLL) file; however, this is not usually necessary because step 3 takes care of the most common country settings.

▶ **To change locale settings on the Macintosh with System 7.1**

1. From the Apple menu, choose Control Panel.
2. Double-click the Date And Time control panel.
3. Specify the date and time settings you want.
4. Close the Date And Time control panel.
5. Double-click the Numbers control panel.
6. Specify the number formats you want.
7. Close the Numbers control panel.

You might want to change other control panel settings as well. For information about changing date, time, and number formats on the Macintosh with other system versions, see your Macintosh documentation.

Writing International Code That Controls Multiple Desktop Applications

If you plan to run Microsoft Excel with another desktop application that supports Visual Basic, you need the appropriate language version of the object libraries for that desktop application. For example, suppose you want to write Italian code that uses features in Microsoft Excel and Microsoft Project. You need the following files:

- The Microsoft Excel Italian object library
- The Visual Basic for Applications Italian object library
- The Microsoft Project Italian object library

For more information about writing code that works with other applications, see Chapter 10, "Controlling and Communicating with Other Applications."

The language version of Microsoft Excel itself and Microsoft Project itself are unimportant. Your Italian code will run on any version of Microsoft Excel and work with any version of Microsoft Project, provided that you include the object libraries in the preceding list.

In the preceding example, if you use, say, Swedish Microsoft Project and you do not have the Microsoft Project Italian object library, you can still run your code if you write it in two languages—Italian for all features and functions that involve

Microsoft Excel, and Swedish for all features and functions that involve Microsoft Project. In such a case, you need separate workbooks for the procedures of each language. You then use the References command on the Tools menu to define a reference between the Swedish and Italian workbooks.

Guidelines for Writing Transportable Visual Basic Code

The following guidelines will help you write your Visual Basic code so it runs more effectively in other locales. Understanding these guidelines is important in order to predict how your code will perform. The guidelines fall into the following areas:

- Processing User-Provided Worksheet Formulas
- Using Conversion Functions
- Displaying Information with Locale-Aware Functions and Statements
- Working with Local Language Text in Your Code
- Other Considerations for Writing Transportable Visual Basic Code

Processing User-Provided Worksheet Formulas

You can use the **Formula** property to enter formulas into cells. If these formulas contain worksheet functions, Microsoft Excel can automatically convert the functions into the appropriate language. If, instead, you write your code so that it uses input boxes to ask the user for similar formulas, Visual Basic can still convert worksheet functions in the provided formulas, but you must use the **FormulaLocal** property instead of the **Formula** property.

```
Cells(1, 1).Formula = "=SUM(A2:A3)"          'Formula will be
                                             'converted automatically
FormulaText = InputBox (EnterFormulaMessage)
Cells(1, 2).FormulaLocal = "=" & FormulaText 'Formula will still be
                                             'converted automatically
```

In the preceding example, FormulaText might also contain something such as `"SUM(A2:A3)"` (if it is translated into English). Entering formulas into cells with the **Formula** or **FormulaLocal** property is the only area in which Visual Basic can convert text from one language into another. But another property, **NumberFormatLocal** (as opposed to **NumberFormat**), does similarly take the user's locale into account. For more information about these properties, search the online Visual Basic Reference for *Formula property, FormulaLocal property, NumberFormat property,* and *NumberFormatLocal property.*

Using Conversion Functions

Visual Basic provides several functions that convert data from one type to another using operating system locale settings. However, some functions are not locale-aware. Whenever possible, use the locale-aware functions to make your code transportable. This is especially important when you're converting from strings to numbers or dates.

For example, the **Str** and **Val** functions always assume a period is the decimal separator, but **CStr**, **CDbl**, **CSng**, **CInt**, and **CLng** use the current operating system settings to determine the decimal separator.

For more information about converting data from one type to another, search the online Visual Basic Reference for *Conversion.*

Special Considerations for Dates and Currencies

Never type dates as strings in your code. Even locale-aware conversion functions can work with strings in ways you might not expect. For example, when the following code is run, it behaves differently in different locales.

```
StartDate = "2/3/94"
NewDate = CDate(StartDate)
```

When run in an English/United States operating system locale, NewDate contains February 3, 1994; in an English/Australia locale, NewDate contains March 2, 1994. This feature is a potential error source when you're programming, but it is very useful for processing user input from a dialog box. **CDate** converts the user's text into the date that the user intends.

However, you, the programmer, should always type dates in code as date literals, such as **#2/3/94#**, so that Visual Basic recognizes exactly the date you intend. For example, in the following code, NewDate contains either February 3, 1994, or March 2, 1994, depending on the locale of the programmer, but not the locale of the user. The date is the same for all users wherever the code is run.

```
StartDate = #2/3/94#
NewDate = CDate(StartDate)
```

Similarly, never type currencies as strings in your code. For example, the following code does not run in any locale except those where the dollar sign ($) is the currency symbol.

```
Money = "$1.22"
NewMoney = CCur(Money)
```

Instead, use decimal numbers, as shown following. (This example assumes the period is the decimal separator in the programmer's locale, but the code runs correctly no matter what the decimal separator is in the user's locale.)

```
Money = 1.22
NewMoney = CCur(Money)
```

Displaying Information with Locale-Aware Functions and Statements

As explained previously, different locales have different conventions for displaying dates, times, numbers, currency, and other information. As a programmer, you can't know the conventions for all your users' locales, but you don't have to know them. Many Visual Basic statements use the user's operating system locale settings to determine the conventions automatically at run time, but some statements are more locale-aware than others. In general, the online Visual Basic Reference tells you whether a statement or function is locale-aware, and if so, in what ways.

For example, the **Print** statement provides little flexibility for different output formats. However, it does use the operating system locale settings. Dates are printed using a correct short date format; numbers are printed with the correct decimal separator; and currencies are printed with the correct symbol.

The **Format** function can accept format codes, but format codes always produce the same type of output without regard to the user's locale. For example, the format code `"mm-dd-yy"` is not appropriate for locales in which the day precedes the month.

For more flexibility, the **Format** function also provides named formats that are locale-aware, including Short Date, Long Date, Long Time, and General Number. Using named formats produces output that is based on the user's operating system locale settings.

```
MyDate = #January 27, 1994#
MyStr = Format(MyDate, "dd-mm-yy")        'Returns 01-27-94
MyStr = Format(MyDate, "Short Date")      'Returns a short date based on
                                          'the locale where the code is
                                          'running
```

The named formats can even generate output in the user's native language, including the names of months and days of the week. This is shown in the illustration in "Running Your Code in Other Language Environments" earlier in this appendix. For more information about printing or formatting in other language environments, search the online Visual Basic Reference for *Print* or *Format.*

Working with Local Language Text in Your Code

Visual Basic includes many features to help your code run properly in other locales, but it cannot automatically translate text from one language to another. For example, you might want your application to include menu text, dialog text, alert messages, and so on. At first, it might seem that you must type this text directly into

your code, which prevents the code from becoming a truly transportable application, but you can replace specific quoted text with variables and references to worksheet names. The cells to which your worksheet names refer can vary depending on the locale in which your code is running. In this way, you can display foreign words and phrases that change depending on your user's locale.

For example, the following worksheet includes a column of names followed by columns of text in three languages.

	A	B	C	D
1	**Localized Strings**	**English**	**French**	**German**
2	Welcome	Welcome to West Coast Sales!	Bienvenue à la société Les Ventes de la Côte Ouest!	Willkommen bei der Nordwind GmbH.
3	CustomerName	What is your name?	Quel est votre nom ?	Wie heißen Sie?
4	AcctNumber	Enter your account number.	Entrez votre numéro de compte.	Bitte geben Sie Ihre Kundennummer ein.
5	NotValidMessage	That transaction is not valid.	Cette transaction n'est pas valide.	Dieser Vorgang kann nicht bearbeitet werden.

To access the specific language text, define the text in the Localized Strings column as worksheet names. These names must refer to the English column, and you can use the **Offset** method to get the text in the other language columns. If you want to work in English, the offset is zero; in French, the offset is one; and in German, the offset is two. You can obtain the offset value in any of the following ways:

- Use the **International** property of the Application object to determine the locale version of Microsoft Excel (xlCountryCode) or the current country setting of the operating system (xlCountrySetting). Make a mathematical adjustment to convert the setting to an offset value, and then define a **Public** variable accordingly. (For more information about **Public** scope, see "Learning More About Visual Basic Programming" in Chapter 6.)
- Display a dialog box asking the user for a language or country name, and then define a **Public** offset variable accordingly.
- Define a **Public** constant that the user sets.

The following code determines an offset code based on the version of Microsoft Excel you're using.

```
Select Case Application.International(xlCountryCode)
    Case 1:                          'US English
        GlobalOffsetCode = 0
    Case 33:                         'French
        GlobalOffsetCode = 1
    Case 49:                         'German
        GlobalOffsetCode = 2
    Case Else:                       'US English as the default language
        GlobalOffsetCode = 0
End Select
```

The following code uses this offset code to display a message box to welcome the user.

```
WelcomeMsg = ThisWorkbook.Worksheets("LocalizationTable").Range _
    ("Welcome").Offset(0, GlobalOffsetCode).Value
MsgBox WelcomeMsg
```

You can use a similar technique to change the text, positions, and sizes of items in dialog boxes, making your code truly transportable.

Other Considerations for Writing Transportable Visual Basic Code

Two considerations for writing code that can vary by locale are how you compare strings and how you work with file input/output.

The rules for comparing strings vary by locale when you're using the less than (<), greater than (>), equal to (=), and **Like** operators, and the **StrComp** function. These differences are explained in the online Visual Basic Reference. For more information, search the online Visual Basic Reference for *Option Compare, Like,* and *StrComp.*

Locale is also a consideration for file input/output. The **Print #** statement puts data into a file as the data is displayed on-screen in a locale-aware format. The **Input #** statement cannot read this information from the file. By comparison, the **Write #** statement puts data into a file in a fixed format, which ensures that the **Input #** statement can read it later in any user locale. For more information about file input/output, search the online Visual Basic Reference for *Print* or *Write.* For more information about file operations in general, search for *Input* and select the topic *Input and Output Keyword Summary.*

APPENDIX B

Switching from the Microsoft Excel 4.0 Macro Language

This appendix introduces users of the Microsoft Excel 4.0 macro language to programming with Visual Basic, the new programming language included with Microsoft Excel version 5.0. In this appendix, you'll learn about how Visual Basic differs from the Microsoft Excel 4.0 macro language, how you can continue using your existing Microsoft Excel 4.0 macros, and where to find more information about Visual Basic.

Visual Basic is a true programming language that features variables with scoping, an integrated editor, and enhanced dialog box and debugging tools. Learning Visual Basic with Microsoft Excel 5.0 makes it easier for you to learn programming with Microsoft Word, Microsoft Project, Microsoft Access, and other Microsoft applications that use Visual Basic. You can also control these other applications easily from within your Visual Basic code. You can use Microsoft Excel with Visual Basic to easily learn Visual Basic programming at your own pace without losing any of your existing investment in Microsoft Excel 4.0 macros.

You can still write and run Microsoft Excel 4.0 macros in Microsoft Excel 5.0. All the new features of Microsoft Excel 5.0 are included in the macro functions. Microsoft Excel 5.0 cannot automatically translate your existing Microsoft Excel 4.0 macros directly into Visual Basic, but you can use your existing macros from your Visual Basic code.

For a complete explanation of Visual Basic in Microsoft Excel, see Chapter 2, "Editing Recorded Macros," and Chapter 5, "Working with Objects in Visual Basic."

Information for Users of Microsoft Excel 4.0 Macros

This section guides experienced users of Microsoft Excel 4.0 macros to information about learning and using Visual Basic. For more detailed information, see the materials cross-referenced in this section.

Recording Macros in Visual Basic

In Microsoft Excel 5.0, you can choose the language in which to record a macro.

▶ **To choose the programming language in which to record a macro**

1. From the Tools menu, choose Record Macro, and then choose Record New Macro.
2. Choose the Options button.
3. Under Language, select either the Visual Basic or the Microsoft Excel 4.0 Macro option button.

In Microsoft Excel 5.0, you can still record your macros in a workbook that opens each time you start Microsoft Excel. In Microsoft Excel version 4.0, this workbook is called GLOBAL.XLM (in Microsoft Excel for Windows) or GLOBAL MACRO SHEET (in Microsoft Excel for the Macintosh). In Microsoft Excel version 5.0, this workbook is now called PERSONAL.XLS or PERSONAL MACRO WORKBOOK. Your existing macros still run; new macros are simply recorded in the new workbook. Microsoft Excel 5.0 creates your new Personal Macro Workbook when you record your first macro.

For more information about recording macros, see Chapter 1, "Automating Repeated Tasks," or search Help for *Recording a macro.*

Acting Directly on Objects in Visual Basic

In Microsoft Excel version 4.0, macros follow the "select, then do" order of actions that pertains to all of Microsoft Excel. With Visual Basic, you don't need to select an object that you want your procedure to change. You can change the object directly.

For example, to make a range of text bold in Microsoft Excel 4.0, you have to first select the range with the SELECT function before changing the format of the text with the FORMAT.FONT function. In Visual Basic, you make a range of text bold by simply setting the **Bold** property of the range to **True.**

```
Range("C1:G5").Font.Bold = True
```

Notice that you can use Visual Basic to directly change an object, in this case the range C1:G5, without selecting it or canceling your current selection. For more information about how to change cells, sheets, charts, shapes, and other objects in Visual Basic, see Chapter 5, "Working with Objects in Visual Basic."

Variables: More Powerful Than Names

In Microsoft Excel version 4.0, to store a value as a variable, you commonly store the value in a name. In Visual Basic, you assign values to variables instead.

Variables are much more flexible than names. You can make variables that are available to all procedures, to just the procedures in a module, or to just a single procedure. You can control the type of data that can be stored in a variable and even create variables that store a combination of data types of your choosing.

In Visual Basic, you can also define constants to hold static values that you need to refer to repeatedly. For more information about variables and constants in Visual Basic, see Chapter 6, "Working with Visual Basic Code in Procedures."

Using Worksheet Functions in a Procedure

Many worksheet functions, except the IF function, can be used directly in Visual Basic procedures. (**If** is a keyword in Visual Basic.) You can use the **Application.** qualifier to run a Microsoft Excel worksheet function rather than a Visual Basic function. For example,

```
X = Acos(-1)
```

causes an error while

```
X = Application.Acos(-1)
```

uses the Microsoft Excel worksheet function ACOS. The only worksheet function that requires you to explicitly specify that you're referring to either the function's Microsoft Excel version or its Visual Basic version is the LOG function because the two function names are spelled the same. The Microsoft Excel LOG function returns the logarithm of a number to the base that you specify. The Visual Basic **Log** function returns the natural logarithm of a specified number.

Using Your Existing Macros in a Procedure

You can include your existing macros in new Visual Basic procedures by using the **Run** method. When you debug Microsoft Excel 4.0 macros as part of your Visual Basic procedures, the Visual Basic debugger steps into your macros as if they were written in Visual Basic. Your macros can return information to a procedure using the RETURN macro function.

For more information and an example of the **Run** method, search the online Visual Basic Reference for *Run method.*

New Tools to Make Debugging Easy

Some of the numerous tools in Visual Basic to help you debug your code are:

- Breakpoints. A *breakpoint* defines a statement or set of conditions at which Visual Basic automatically stops your program and puts the code in break mode without running the statement containing the breakpoint. You can use breakpoints to examine your code and variables at places in your procedures where you think errors are occurring.
- Debugging buttons. The debugging buttons are included on the Visual Basic toolbar, which appears whenever a Visual Basic module is active. You can use these buttons to run your code one statement or procedure at a time, to control breakpoints, and to examine values of variables and expressions.
- The Debug window. You can use the Step Into command on the Run menu to display the Debug window. With the Debug window, you can monitor the values of expressions and variables while running your code one statement at a time. You can also use this window while running your code to change the value of variables and properties to see how different values affect your code.

The Debug window contains three areas:

- The Immediate pane. This pane displays results from debugging statements in your code. You can also request debugging information by typing commands directly in the Immediate pane.
- The Watch pane. This pane displays expressions whose values you have decided to monitor as your code runs.
- The Code pane. This pane displays the most recently executed statement and allows browsing of surrounding pieces of code.

For more information about these tools and other aspects of debugging in Microsoft Excel 5.0, see Chapter 8, "Testing and Debugging Your Code."

Visual Basic Equivalents for Common Macro Functions

The easiest way to see the Visual Basic equivalents for common macro functions and Microsoft Excel commands is to use the Macro Recorder to record macros in Visual Basic. You can open two windows on your workbook, with one window showing your Visual Basic module and the other showing the worksheet or chart that you're working on while recording. As you work, Microsoft Excel adds Visual Basic statements to your module.

No matter how you write your programs in Microsoft Excel, there are common tasks that you'll want to accomplish, such as selecting ranges, controlling how macros run, finding information in workbooks, and getting information about workbooks and objects. The following table shows you where to look in this book for information about how to accomplish these tasks with Visual Basic.

For information about	See this chapter
Selecting cells and ranges on worksheets	Chapter 5, "Working with Objects in Visual Basic"
Searching for text and values in workbooks	
Receiving information about objects in Microsoft Excel	
Manipulating references	
Controlling how statements in a procedure run	Chapter 7, "Controlling How Your Code Runs"
Running procedures in response to events	Chapter 13, "Creating Automatic Procedures and Add-in Applications"

Microsoft Excel 5.0 also includes example code written in both Visual Basic and the Microsoft Excel 4.0 macro language. To see this code, open the workbooks in the EXAMPLES directory or folder in the directory or folder where you installed Microsoft Excel 5.0.

Creating Custom Commands and Dialog Boxes with Visual Basic

Microsoft Excel 5.0 includes new, advanced tools for creating custom menus, commands, and dialog boxes. For more information about creating custom commands and dialog boxes using Visual Basic, see Chapter 11, "Controls and Dialog Boxes," and Chapter 12, "Menus and Toolbars."

Creating Custom Commands

To create a custom menu or command with Microsoft Excel version 4.0, you first create a menu or command table. You then use macro functions such as ADD.MENU and ADD.COMMAND to place your custom menu or command on a menu or menu bar.

To create a custom menu or command with Microsoft Excel version 5.0, you use the Menu Editor to assign custom commands and menus to menu bars. For more information about using the Menu Editor, see "Modifying the Menuing System with the Menu Editor" in Chapter 12.

Displaying Built-in Dialog Boxes

In Microsoft Excel version 4.0, to display built-in dialog boxes when running a macro, you use the question-mark form of the macro function corresponding to the dialog box. For example, the DEFINE.STYLE? macro function displays the dialog box used to define worksheet styles.

In Microsoft Excel version 5.0, to display built-in dialog boxes while running a procedure, you use the **Dialogs** method with the identifier of the dialog box you want displayed. For example, this statement displays the dialog box for the Open command on the File menu.

```
Application.Dialogs(xlDialogOpen).Show
```

Creating and Displaying Custom Dialog Boxes

To create a custom dialog box with Microsoft Excel version 4.0, you use the Dialog Editor to generate a dialog box definition that you place on a macro sheet. You then use the DIALOG.BOX macro function to display your dialog box.

In Microsoft Excel version 5.0, custom dialog boxes are stored on dialog sheets in your workbook. To create a custom dialog box, you use the buttons on the Forms toolbar to create and run a dialog box on a dialog sheet in a workbook. You use the **Show** method in a Visual Basic procedure to display your custom dialog box.

You can use the **DialogBox** method in your Visual Basic procedures to run a Microsoft Excel 4.0 custom dialog box. The following example uses the **DialogBox** method to display such a dialog box and then tests the result. The variable `DialogRange` refers to the range on a Microsoft Excel 4.0 macro sheet that contains the dialog-box definition table.

```
Result = DialogRange.DialogBox
If Not Result Then
    ' User canceled the dialog box
Else
    ' Result is position number of chosen control
End If
```

For more information about using dialog boxes in programming Microsoft Excel 5.0, see Chapter 11, "Controls and Dialog Boxes."

Creating Add-in Applications

In Microsoft Excel version 4.0, all you need to do to convert a macro to an add-in macro is save the macro sheet in the add-in file format, and it will be hidden from view. Microsoft Excel 4.0 add-in macros still run in Microsoft Excel version 5.0. (Add-in macros are called add-in applications in Microsoft Excel 5.0.)

To create an add-in application in Microsoft Excel version 5.0, you use the Make Add-In command on the Tools menu when a Visual Basic module, Microsoft Excel 4.0 macro sheet, or dialog sheet is active. However, before you convert a workbook containing Visual Basic code to an add-in application, you must save your code in a workbook with a different filename if you want to continue working with the code. Once you convert it to an add-in application, any code in the workbook cannot be edited again. You can distribute your add-in application to other Microsoft Excel 5.0 users, who can then run your add-in application but cannot see or change your code.

For more information about add-in applications in Microsoft Excel 5.0, see Chapter 13, "Creating Automatic Procedures and Add-in Applications."

Macros That Call Microsoft Excel 4.0 Add-ins

Several Microsoft Excel version 4.0 add-ins are not available with Microsoft Excel version 5.0. Their functionality has been incorporated directly into Microsoft Excel. Most of your Microsoft Excel version 4.0 macros should run properly in Microsoft Excel version 5.0 without modification, because Microsoft Excel translates their function calls automatically.

If you have macros that include functions from the Analysis ToolPak (ANALYSIS.XLA and ANALYSF.XLA), Scenario Manager (SCENARIO.XLA) or Add-In Functions (ADDINFNS.XLA) add-ins, however, you should use the Link Changer included with Microsoft Excel 5.0 to update the references to these add-ins.

For information about the Link Changer, see "Changing Links to Microsoft Excel version 4.0 add-ins" in the Microsoft Excel Macro Functions Help.

APPENDIX C

Displaying Custom Help Topics

With Microsoft Excel, you can create a Help system to provide users with information about dialog boxes, commands, menus, buttons, and messages displayed in the course of running a macro.

The Help topics you provide can include powerful and visually interesting features, such as pop-up windows, graphics, the ability to jump from one topic to another, *hypergraphics* (graphics with one or more invisible graphic objects, or *hot spots,* assigned to them), and much more.

To create and compile Help files, you need the Microsoft Windows Help Compiler version 3.1 or later. For more information about the Help compiler and how to obtain it, see "Getting More Information About the Microsoft Windows Help Compiler" later in this appendix.

The Help compiler creates Help files for both Windows-based and Macintosh applications. If you're using Microsoft Excel for the Macintosh, you first create and compile the source files on a Microsoft Windows–compatible computer and then transfer the compiled files to the Macintosh.

The Microsoft Windows Help Compiler (HC31.EXE) creates Help files; it does not display them. To display Help files, you must use one of the two Help *viewers*—Microsoft Windows Help (WINHELP.EXE) or Microsoft Help for the Macintosh, depending on the operating environment in which your Help system will be used.

Microsoft Windows Help is included with Microsoft Windows and is usually located in your WINDOWS directory.

Microsoft Help for the Macintosh is included with Microsoft Excel and is usually located in the MICROSOFT folder within the SYSTEM FOLDER. You must purchase the Help compiler separately. For more information about obtaining the Help compiler, see "Getting More Information About the Microsoft Windows Help Compiler" later in this appendix.

Creating a Help System

After you've reviewed how to create Help topics, you're ready to create the Help system for your application in Microsoft Excel. The following list shows the general steps. For specific information about each step, see your Help compiler documentation. This section assumes you are using a Microsoft Windows–compatible computer.

1. After you have created the custom commands, dialog boxes, and the other elements of your application, gather information for the corresponding custom Help topics. Decide which items need custom Help, what type of Help to provide, how to structure your Help files, and how to structure Help topics within each file.
2. Write the text for the Help topics in the source files, enter all required control codes, and save the source files in rich-text format (RTF).
3. Create a project file, and then compile the Help file.
4. Start Microsoft Excel, run your custom application, and test the display of your custom Help topics.

Displaying Help Topics

Microsoft Windows Help displays only files that have been compiled by the Microsoft Windows Help Compiler. Microsoft Help for the Macintosh displays both compiled files and plain text files.

To display a Help topic in Microsoft Excel, use one of the following:

- The **Help** method of the Application object
- The Help button in a message displayed by the **MsgBox** or **InputBox** statement
- A Help button in a custom dialog box

Using the Help Method

When you run the **Help** method in your Visual Basic code, Microsoft Excel displays the specified Help topic. The **Help** method has the following syntax:

object.**Help** ***"helpFile", helpContextID***

Here, *object* is the Application object name, *helpFile* is the name of the file containing the topic and can include the full path, and *helpContextID* identifies the topic within the Help file.

For example, in Microsoft Excel for Windows, you might use the following Visual Basic code to display Help topic number 103.

```
Application.Help "C:\REPORTS\REPORTS.HLP", 103
```

In Microsoft Excel for the Macintosh, you might use this Visual Basic code to display the same topic.

```
Application.Help "Hard Disk:Reports:Reports Help", 103
```

Custom Help for Dialog Boxes

You can use the **HelpButton** property of buttons in combination with the **Help** method of the Application object to display Help for dialog boxes.

Each button in a dialog box has a **HelpButton** property. If the **HelpButton** property of any button is **True**, then pressing F1 in Microsoft Excel for Windows (or COMMAND+SHIFT+ QUESTION MARK in Microsoft Excel for the Macintosh) when the dialog box is displayed causes Microsoft Excel to respond as if you had chosen the button. Microsoft Excel then runs the Visual Basic code attached to the button.

If you want the attached Visual Basic code to display a Help topic, use the **Help** method of the Application object as described in the previous section. Only one button in a dialog box can have its **HelpButton** property set to **True**, so you usually create a button exclusively for displaying Help, with a caption that reads "Help." If no button has its **HelpButton** property set to **True**, pressing F1 does nothing.

Getting More Information About the Microsoft Windows Help Compiler

You can obtain the Microsoft Windows Help Compiler and information about how to use it from any of the following sources:

- *The Microsoft Windows Help Authoring Guide* is available on CD-ROM through the Microsoft Software Developer Network. This comprehensive guide explains how to design and implement a complete Help system and describes the Microsoft Windows Help Compiler. The Help compiler and sample code is included with the guide. To obtain the *Microsoft Windows Help Authoring Guide*, contact your software vendor or Microsoft Corporation.
- Microsoft Visual Basic, Professional Edition. This version of Visual Basic includes the Help compiler and documentation explaining how to use it. The documentation is a condensed version of that found in the *Microsoft Windows Help Authoring Guide* but is thorough and should meet the needs of most users.
- The Microsoft Excel Developer's Kit and the Microsoft Windows Software Development Kit. These kits include the Help compiler. The accompanying documentation assumes you know how to author Help using previous versions of the Help compiler. To obtain either kit, contact your software vendor or Microsoft Corporation.

APPENDIX D

Toolbar Buttons in Microsoft Excel

Microsoft Excel provides more than 200 toolbar buttons and boxes that you can display on your toolbars. Most of these are initially displayed on one of the built-in toolbars. The tables in this appendix provide brief descriptions of all the toolbar buttons and boxes grouped by the category of task with which they are associated. Each button or box is listed with an identifying number for use with Visual Basic and the Microsoft Excel 4.0 macro language.

For more detailed information about a button or box, click the Help button on the Standard toolbar, and then on the appropriate toolbar click the item about which you want more information. If the button or box is not displayed on a toolbar, double-click the Help button on the Standard toolbar, and then search for the item's name in Help.

Auditing Category

The following table describes the actions performed by the toolbar buttons in the Auditing category.

Button		Action	Number
	Attach Note	Adds a note to the current selection.	154
	Remove All Arrows	Removes all arrows used in auditing.	153
	Remove Dependent Arrows	Removes one level of arrows tracing dependent cells.	147
	Remove Precedent Arrows	Removes one level of arrows tracing precedent cells.	149
	Show Info Window	Displays the Info window.	243
	Trace Dependents	Shows the cells that depend on the current cell.	148

Button		Action	Number
	Trace Error	Shows cells causing the error in the current cell.	174
	Trace Precedents	Shows the cells that the current cell depends on.	242

Charting Category

The following table describes the actions performed by the toolbar buttons and box in the Charting category.

Button		Action	Number
	3-D Area Chart AutoFormat	Creates a 3-D area chart with 3-D markers.	109
	3-D Bar Chart AutoFormat	Creates a bar chart with 3-D markers.	110
	3-D Column Chart AutoFormat	Creates a column chart with 3-D markers.	111
	3-D Line Chart AutoFormat	Creates a 3-D line or ribbon chart.	113
	3-D Perspective Column Chart AutoFormat	Creates a 3-D column chart with a 3-D plot area.	112
	3-D Pie Chart AutoFormat	Creates a 3-D pie chart with value labels expressed as percentages.	114
	3-D Surface Chart AutoFormat	Creates a 3-D surface chart.	116
	Area Chart AutoFormat	Creates a simple area chart.	103
	Bar Chart AutoFormat	Creates a simple bar chart.	104
	Chart Type	Displays the Chart Type palette so that you can change the type of a chart.	234
	ChartWizard	Starts the ChartWizard so that you can edit the selected chart or create an embedded chart.	121
	Column Chart AutoFormat	Creates a simple column chart.	105
	Default Chart	Creates a chart using the default chart type set on the Chart tab of the Options dialog box.	120

Button		Action	Number
	Doughnut Chart AutoFormat	Creates a doughnut chart.	145
	Horizontal Gridlines	Adds or deletes major gridlines for the value axis.	122
	Legend	Adds or deletes a chart legend.	124
	Line Chart AutoFormat	Creates a line chart with markers.	107
	Line/Column Chart AutoFormat	Creates a combination chart with a column chart overlaid by a line chart.	118
	Pie Chart AutoFormat	Creates a pie chart with value labels expressed as percentages.	108
	Radar Chart AutoFormat	Creates a radar chart with markers.	117
	Stacked Column Chart AutoFormat	Creates a stacked column chart.	106
	Vertical Gridlines	Adds or deletes major gridlines for the category axis.	123
	Volume/High-Low-Close Chart AutoFormat	Creates a combination chart with a column chart overlaid by a line chart containing three data series for high, low, and closing stock prices.	119
	XY (Scatter) Chart AutoFormat	Creates an xy (scatter) chart with markers but no lines.	115

Custom Category

The following table describes and gives identifying numbers for the buttons available in the Custom category. None of these buttons has a built-in action. You can use these buttons with your own procedures or Microsoft Excel 4.0 macros.

Button	Button number	Button	Button number
Balloon	219	Mug	218
Bell	226	Note	210
Blank	231	Notebook	217
Calculator	237	Open	206
Camera	228	Palette	97
Clock	213	Save	207
Club	224	Smile	211
Diamond	222	Solitaire	201

Button	Button number	Button	Button number
Envelope	230	Spade	223
Fish	214	Speaker	209
Frown	212	Tack	215
Hand	229	Telephone	220
Heart	221	Trash	225
Integral	200	Undo	208

Data Category

The following table describes the actions performed by the toolbar buttons in the Data category.

Button		Action	Number
	AutoFilter	Displays only those rows that match the value contained in the active cell and inserts drop-down arrows next to each column label in your list.	168
	Group	Defines the selected detail rows or columns as a group. In a pivot table, groups items by category to create a single item from multiple items.	130
	Hide Detail	Hides details of the selection.	175
	PivotTable Field	Inserts a subtotal for a row field or column field in an existing pivot table. If a cell in the data area is selected, clicking this button defines the summary functions used in the selected data field.	171
	PivotTable Wizard	Starts the PivotTable Wizard, which guides you through all the steps needed to create or modify a pivot table.	167
	Refresh Data	Updates the data in a pivot table after the source data is changed.	170
	Show Detail	Shows details of the selection.	173
	Show Pages	Copies each page of a page field to a new worksheet in the current workbook.	172

Button		Action	Number
	Sort Ascending	Sorts the current list in order from lowest value to highest value, using the column that contains the active cell.	134
	Sort Descending	Sorts the current list in order from highest value to lowest value, using the column that contains the active cell.	135
	Ungroup	Removes selected rows or columns from their group. In a pivot table, separates a combined group of items, replacing each instance of the group with all the items it contained.	129

Drawing Category

The following table describes the actions performed by the toolbar buttons and boxes in the Drawing category.

Button		Action	Number
	Arc	Displays a cross-hair pointer that you use to draw an arc or circle segment.	85
	Arrow	Displays a cross-hair pointer that you use to draw an arrow on the active worksheet or adds an arrow to the active chart document.	77
	Bring To Front	Places the selected objects in front of all other objects.	95
	Color	Displays the color palette that you use to assign a color to the selected object.	233
	Create Button	Displays a cross-hair pointer that you use to draw a button to which you assign a procedure or Microsoft Excel 4.0 macro.	80
	Dark Shading	Applies dark shading to selected cells or objects.	49
	Drawing	Displays or hides the Drawing toolbar.	240
	Drawing Selection	Displays a cross-hair pointer that you use to select drawing objects.	184
	Drop Shadow	Adds a shadowed rectangle around the selected cells or adds a shadow to the border of the selected objects.	51

Button or box		Action	Number
	Ellipse	Displays a cross-hair pointer that you use to draw an ellipse or circle.	84
	Filled Arc	Displays a cross-hair pointer that you use to draw an arc or circle segment filled with the window background pattern and color.	90
	Filled Ellipse	Displays a cross-hair pointer that you use to draw an ellipse or circle filled with the window background pattern and color.	89
	Filled Freeform	Displays a cross-hair pointer that you use to draw a polygon that is a combination of freehand and straight lines filled with the window background pattern and color.	92
	Filled Polygon	Displays a cross-hair pointer that you use to draw a polygon filled with the window background pattern and color.	91
	Filled Rectangle	Displays a cross-hair pointer that you use to draw a rectangle or square filled with the window background pattern and color.	88
	Freehand	Displays a cross-hair pointer that you use to draw a continuous freehand line.	78
	Freeform	Displays a cross-hair pointer that you use to draw a shape that is a combination of freehand and straight lines.	87
	Group Objects	Joins the selected graphic objects into a single graphic object.	93
	Light Shading	Applies light shading to selected cells or objects.	50
	Line	Displays a cross-hair pointer that you use to draw a straight line.	76
	Pattern	Changes the pattern and pattern color of filled objects.	232
	Polygon	Displays a cross-hair pointer that you use to draw a polygon.	86
	Rectangle	Displays a cross-hair pointer that you use to draw a rectangle or square.	83
	Reshape	Changes the shape of a polygon.	82
	Selection	Selects graphic objects by drawing a selection rectangle around them.	81
	Send To Back	Places the selected objects behind all other objects.	96

Button or box		Action	Number
	Shape	Displays a palette of drawing shapes.	235
	Text Box	Displays a cross-hair pointer that you use to draw a text box, in which you can type text.	79
	Ungroup Objects	Separates grouped objects into individual objects.	94

Edit Category

The following table describes the actions performed by the toolbar buttons in the Edit category.

Button		Action	Number
	Clear Contents	Removes only the formulas or values from the selected cells or deletes the selected objects.	15
	Clear Formats	Removes only the formats from the selected cells or objects.	16
	Copy	Copies the selection to the Clipboard.	13
	Cut	Cuts the selection and places it on the Clipboard.	12
	Delete	Removes the selected cells or objects. If cells are deleted, using the Delete button shifts surrounding cells to fill in the space.	19
	Delete Column	Deletes the selected column.	21
	Delete Row	Deletes the selected row.	20
	Fill Down	Copies the values, formulas, and formats of the cells in the top row of the selection into the cells below. To copy from the bottom row upward, hold down SHIFT and click this button.	26
	Fill Right	Copies the values, formulas, and formats of the cells in the left column of the selection into the cells to the right. To copy from the right column to the left, hold down SHIFT and click this button.	25
	Format Painter	Copies formats out of one selection and into another.	185
	Insert	Inserts blank cells in place of the current selection.	22

Button		Action	Number
	Insert Column	Inserts entire columns in the location of the current selection.	24
	Insert Row	Inserts entire rows in the location of the current selection.	23
	Paste	Pastes the contents of the Clipboard.	14
	Paste Formats	Pastes only the formats from the copied cells into the selected cells.	17
	Paste Values	Pastes only the values from the copied cells into the selected cells.	18
	Repeat	Repeats the last action or command.	11
	Undo	Undoes the last action or command.	10

File Category

The following table describes the actions performed by the toolbar buttons in the File category.

Button		Action	Number
	Find File	Displays the Find File dialog box so that you can search for files by attributes.	177
	Insert Chart Sheet	Inserts a new chart sheet into the active workbook.	8
	Insert Dialog	Inserts a new dialog sheet to the left of the selected sheet.	245
	Insert MS Excel 4.0 Macro	Inserts a new Microsoft Excel 4.0 macro sheet into the active workbook.	6
	Insert Module	Inserts a new module into the active workbook.	190
	Insert Worksheet	Inserts a new worksheet into the active workbook.	7
	New Workbook	Creates a new workbook.	9
	Open	Displays the Open dialog box so that you can open an existing document.	1
	Print	Prints the active sheet according to the current page setup and print settings.	3
	Print Preview	Displays the active sheet in print preview.	4

Button		Action	Number
	Routing Slip	Edits the routing slip for a document. Adds a routing slip if no routing slip exists.	162
	Save	Saves changes made to the active workbook.	2
	Send Mail	Sends the current document as an electronic mail message.	163
	Set Print Area	Defines the selected range of cells as the area to print.	5
	Toggle File Status	Switches a file between read-only and read-write status.	165
	Update File	Updates a read-only file to the last saved version.	164

Formatting Category

The following table describes the actions performed by the toolbar buttons and boxes in the Formatting category.

Button or box		Action	Number
	AutoFormat	Applies the last table format applied with the AutoFormat command on the Format menu. To apply the next format in the list, hold down SHIFT and click this button.	52
	Borders	Displays a palette of various border types.	198
	Bottom Border	Adds or deletes a border along the bottom edge of each selected cell.	47
	Bottom Double Border	Adds or deletes a double border along the bottom edge of each selected cell.	48
,	Comma Style	Applies the comma style to the selected cells.	55
$	Currency Style	Applies the currency style to the selected cells.	53
	Dark Shading	Applies a dark shading to the selected cells or objects.	49

Button or box		Action	Number
	Decrease Decimal	Displays one less decimal place each time you click the button.	57
	Increase Decimal	Displays an additional decimal place each time you click the button.	56
	Left Border	Adds or deletes a border along the left edge of each selected cell.	44
	Light Shading	Applies a light shading to the selected cells or objects.	50
	Outline Border	Adds a border around the outermost edges of the selected cells.	43
	Percent Style	Applies the percent style to the selected cells.	54
	Right Border	Adds or deletes a border along the right edge of each selected cell.	45
Normal	Style	Lists the defined cell styles. Select the style you want to apply to the selection, or type a new name and press ENTER to define a style based on the formats of the selected cells.	70
	Top Border	Adds or deletes a border along the top edge of each selected cell.	46

Forms Category

The following table describes the actions performed by the toolbar buttons in the Forms category.

Button		Action	Number
	Check Box	Creates a check box control.	140
	Combination Drop-Down Edit	Creates a combination drop-down box and edit box.	197
	Combination List-Edit	Creates a combination list box and edit box.	188
	Control Properties	Changes properties for the selected control.	238

Button		Action	Number
	Drop-Down	Creates a drop-down list.	182
	Edit Box	Creates an edit box.	142
	Edit Code	Edits or creates code for the selected control.	244
	Group Box	Creates a group box.	181
	Label	Creates a text label.	199
	List Box	Creates a list box.	144
	Option Button	Creates an option button control.	141
	Run Dialog	Runs your dialog box in a separate window.	187
	Scroll Bar	Creates a scroll bar control.	143
	Spinner	Creates a spinner control.	183
	Tab Order	Sets the tab order for controls in a dialog sheet.	186
	Toggle Grid	Displays or hides the grid on a dialog sheet.	239

Formula Category

The following table describes the actions performed by the toolbar buttons in the Formula category.

Button		Action	Number
	AutoSum	Inserts the SUM function and a proposed sum range based on the data above or to the left of the active cell.	39
	Colon	Inserts a colon (:) in the formula bar at the location of the insertion point.	35
	Comma	Inserts a comma (,) in the formula bar at the location of the insertion point.	36

Button		Action	Number
	Constrain Numeric	Restricts handwriting recognition to digits and punctuation (for use with Microsoft Windows for Pen Computing).	42
"/"	Division Sign	Inserts a division sign (/) in the formula bar at the location of the insertion point.	31
"$"	Dollar Sign	Inserts a dollar sign ($) in the formula bar at the location of the insertion point.	38
"="	Equal Sign	Inserts an equal sign (=) in the formula bar at the location of the insertion point.	27
"^"	Exponentiation Sign	Inserts an exponentiation sign (^) in the formula bar at the location of the insertion point.	32
	Function Wizard	Displays the Function Wizard so that you can insert a function into the selected cells.	40
"("	Left Parenthesis	Inserts an opening parenthesis [(] in the formula bar at the location of the insertion point.	33
"–"	Minus Sign	Inserts a minus sign (–) in the formula bar at the location of the insertion point.	29
"*"	Multiplication Sign	Inserts a multiplication sign (*) in the formula bar at the location of the insertion point.	30
=ab	Paste Names	Displays the Paste Names dialog box so that you can paste worksheet names into the selected cells.	41
"%"	Percent Sign	Inserts a percent sign (%) in the formula bar at the location of the insertion point.	37
"+"	Plus Sign	Inserts a plus sign (+) in the formula bar at the location of the insertion point.	28
")"	Right Parenthesis	Inserts a closing parenthesis [)] in the formula bar at the location of the insertion point.	34

Macro Category

The following table describes the actions performed by the toolbar buttons in the Macro category.

Button		Action	Number
	Function Wizard	Displays the Function Wizard so that you can insert a function into the selected cells.	40
	Insert Module	Inserts a new module into the active workbook.	190
	Instant Watch	Displays the value of the selected expression.	194
	Menu Editor	Displays the Menu Editor dialog box so that you can change menu bars and commands.	192
	Object Browser	Displays procedures, objects, methods, and properties.	191
	Paste Names	Displays the Paste Names dialog box so that you can paste worksheet names into the selected cells.	41
	Record Macro	Records actions to create a procedure.	98
	Resume Macro	Resumes running a paused procedure.	102
	Run Macro	Runs the procedure you are editing, starting at the insertion point.	100
	Step Into	Executes the next statement by stepping into the procedure.	195
	Step Macro	Runs the procedure you are editing one statement at a time, starting at the insertion point.	101
	Step Over	Runs the next statement by stepping over the procedure.	196
	Stop Recording Macro	Stops the currently running procedure. If the Macro Recorder is running, clicking this button stops the Macro Recorder.	99
	Toggle Breakpoint	Sets or clears breakpoints.	193

Text Formatting Category

The following table describes the actions performed by the toolbar buttons and boxes in the Text Formatting category.

Button or box		Action	Number
	Align Left	Aligns the cell, text box, button contents, or selected text on a chart to the left.	63
	Align Right	Aligns the cell, text box, button contents, or selected text on a chart to the right.	65
	Bold	Applies bold formatting to the selected cells or objects.	58
	Center	Centers the cell, text box, button contents, or selected text on a chart.	64
	Center Across Columns	Centers the text from one cell horizontally across the selected columns.	67
	Cycle Font Color	Changes the color of fonts in selected objects.	62
	Decrease Font Size	Decreases the font size of the selected text to the next smaller size in the Font Size box.	72
	Double Underline	Applies double underlining to the selected cells or objects.	176
	Font Color	Displays a palette of colors you can use to change the font color in selected cells, or of selected text.	236
Arial	Font	Lists the available fonts. Select the font you want to apply to the current selection.	68
10	Font Size	Lists the available sizes for the font shown in the Font box. Select the size you want to apply to the current selection.	69

Button or box		Action	Number
	Increase Font Size	Increases the font size of the selected text to the next larger size in the Font Size box.	71
	Italic	Applies italic formatting to the selected cells or objects.	59
	Justify Align	Increases word spacing in a text box, button, or cell with wrapped text to fill out each line.	66
	Rotate Text Down	Rotates text sideways so that it reads from top to bottom.	75
	Rotate Text Up	Rotates text sideways so that it reads from bottom to top.	74
	Strikethrough	Applies strikethrough formatting to the selected cells or objects.	61
Normal	Style	Lists the defined cell styles. Select the style you want to apply to the current selection.	70
	Underline	Applies underlining to the selected cells or objects.	60
	Vertical Text	Places text characters upright, one below another, so that the text reads vertically.	73

TipWizard Category

The following table describes the actions performed by the toolbar buttons and box in the TipWizard category.

Button or box		Action	Number
	Tip Help	Gives an in-depth explanation of the current tip.	180
TipWizard Box	TipWizard box	Displays the current tip. Using this box, you can scroll through a list of tips.	178
	TipWizard	Indicates when a tip is available and displays the TipWizard toolbar.	179

Utility Category

The following table describes the actions performed by the toolbar buttons and boxes in the Utility category.

Button or box		Action	Number
	AutoSum	Inserts the SUM function and a proposed sum range based on the data above or to the left of the active cell.	39
	Calculate Now	Calculates the formulas in all the open documents or the formula in the formula bar.	126
	Camera	Copies a picture of the selected range of cells. Click in the worksheet to paste the picture and link the picture to the source cells.	125
	Create Button	Displays a cross-hair pointer that you use to draw a button to which you assign a procedure or Microsoft Excel 4.0 macro.	80
	Freeze Panes	Freezes or unfreezes the split in the active window; if no split exists, splits the window above and to the left of the active cell and freezes panes.	137
	Full Screen	Toggles Full Screen display.	241
	Function Wizard	Displays the Function Wizard so that you can insert a function into the selected cells.	40
	Help	Adds a question mark to the mouse pointer. When you choose a command or click a screen region, a Help topic on that item is displayed. Click this button again to cancel the question mark. Double-click this button to display Help's Search dialog box.	128
	Lock Cell	Formats the select cells or objects as locked so that no changes can be made to them when the document is protected.	136
	Microsoft Access	Starts Microsoft Access, or switches to it if it is already running.	216

Button or box		Action	Number
	Microsoft FoxPro	Starts Microsoft FoxPro, or switches to it if it is already running.	160
	Microsoft Mail	Starts Microsoft Mail, or switches to it if it is already running.	203
	Microsoft PowerPoint	Starts Microsoft PowerPoint, or switches to it if it is already running.	204
	Microsoft Project	Starts Microsoft Project, or switches to it if it is already running.	205
	Microsoft Schedule+	Starts Microsoft Schedule+, or switches to it if it is already running.	227
	Microsoft Word	Starts Microsoft Word, or switches to it if it is already running.	202
=ab	Paste Names	Displays the Paste Names dialog box so that you can paste worksheet names into the selected cells.	41
	Scenarios	Adds, displays, or edits scenarios.	166
	Select Current Region	Selects a rectangular range of cells around the active cell that is bounded by blank rows and columns.	133
	Select Visible Cells	Selects only the visible cells in a selection and does not include hidden rows or columns.	132
	Show Outline Symbols	Displays or hides the outlining symbols on a worksheet containing an outline; if no outline exists, displays a message asking if you want to create one.	131
	Sort Ascending	Sorts the current list in order from lowest value to highest value, using the column that contains the active cell.	134
	Sort Descending	Sorts the current list in order from highest value to lowest value, using the column that contains the active cell.	135
	Spelling	Starts checking the spelling of the active document or the text in the formula bar.	127
	Text Box	Displays a cross-hair pointer that you use to draw a text box in which you can type text.	79

Button or box		Action	Number
100%	Zoom Control	Changes the magnification of the active worksheet.	189
	Zoom In	Displays the worksheet at the next higher magnification.	138
	Zoom Out	Displays the worksheet at the next lower magnification.	139

Index

A

B

C

D

E

F

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Getting Help in a Visual Basic

To get Help about a property, module, statement, or other keyword, select the keyword, and then press F1 (Microsoft® Excel for Windows™) or ⌘+ / (Microsoft Excel for the Macintosh®).


Object Browser

To browse the object libraries available to Visual Basic, click the Object Browser button.

Automating Tasks with the Macro Recorder

You can quickly automate tasks and create custom commands using the *Macro Recorder*. When you start the Macro Record Microsoft Excel records your keystrokes and mouse actions so that you can play them back later with a single keystroke c a push of a button.

To record a macro


Record Macro

1. From the Tools menu, choose Record Macro, and then choose Record New Macro.
   Or click the Record Macro button on the Visual Basic toolbar.
2. Type a name and a description of the macro, and then choose the OK button.


Stop Macro

3. Carry out the actions you want to record.
4. Click the Stop Macro button.

After recording a macro, you can run it using the Macro command on the Tools menu, or you can assign it to a menu command, button, or other object.

Keyboard Shortcuts

To	Press (Windows)	Press (Macintosh)
Run a Visual Basic procedure from the insertion point	F5	⌘+R or F5
Interrupt a macro	ESC	⌘+PERIOD
Display the Object Browser dialog box	F2	⌘+B
Display the Debug window	CTRL+G	⌘+G
Run the next line of code in a Visual Basic procedure from the Debug window	F8	⌘+I or F8

Microsof