EXTENDED LEARNING MODULE M

PROGRAMMING IN EXCEL WITH VBA

Student Learning Outcomes

1. Explain the value of using VBA with Excel.
2. Define a macro.
3. Build a simple macro using a Sub procedure and a Function procedure.
4. Describe an object.
5. Explain the difference between a comment, a variable, and a constant.
6. List the various Visual Basic Application data types and operators.
7. Describe and build a macro that uses the *If-Then-Else, For-Next, Do-Until, Do-While,* and *Select Case* structures.

Introduction

VBA, which stands for *Visual Basic for Applications,* is a programming language developed by Microsoft. Excel, along with the other members of Microsoft Office, includes the VBA language. VBA extends and customizes Excel, allowing you to do things that Excel itself is not able to do, such as calculating values in cells based on user input values.

Excel VBA is a programming application that allows you to use Visual Basic code to run the many features of Excel, thereby allowing you to customize your Excel applications. Units of VBA code are often referred to as *macros.* Although this module covers more formal terminology, the term macro will be used as a general way to refer to any VBA code. It is not essential to make a further software purchase in order to learn the elements of Visual Basic programming. Excel VBA comes free with Excel (97, 2000, 2002, 2003, etc.) and provides a great deal of the functionality (which happens to be the topic of this module).

One of the great advantages of Excel VBA is the macro recorder. The *macro recorder* is a software tool that lets you record a sequence of commands in Excel and save them as a macro. This is invaluable if you are struggling with some programming syntax. Just get the recorder to do it for you and then view the code to see how it is done.

Why VBA?

LEARNING OUTCOME 1

Excel programming terminology can be a bit confusing. For example, VBA is a programming language, but it also serves as a macro language. A ***macro language*** is a programming language that includes built-in commands that mimic the functionality available from menus and dialog boxes within an application. What do you call something written in VBA and executed in Excel? Is it a macro or is it a program? Excel's Help system often refers to VBA procedures as macros. A ***macro*** is a set of actions recorded or written by a user. For instance, you can create a macro that always prints your name in bold on a spreadsheet. You can name it PrintName and then reuse the PrintName macro in any spreadsheet. This term means that a series of steps is completed automatically. For example, if you write a macro that adds a background color to some cells and then prints the worksheet, you have automated those two steps.

LEARNING OUTCOME 2

People use Excel for thousands of different tasks, such as:

- Budgeting and forecasting
- Analyzing data
- Creating invoices and other forms
- Developing charts from data

One thing virtually every user has in common is the need to automate some aspect of Excel. For example, you might create a VBA macro to format and print a month-end sales report. You can execute the macro with a single command, triggering Excel to automatically perform many time-consuming tasks, such as inserting your name at the top of every spreadsheet you create, specifying which rows are to be repeated at the top of each printed page, adding information in the footer area, and so on.

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

If you are able to perform an operation manually, you can use the macro recorder to capture that operation. This is a very quick and easy process and requires no prior knowledge of the VBA language.

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

Here are some common uses for VBA macros:

- *Inserting text*—If you often need to enter your name into worksheets, you can create a macro to do the typing for you.
- *Automating a task*—If you're a sales manager and need to prepare a month-end sales report, you can develop a VBA macro to do it for you.
- *Automating repetitive tasks*—If you need to perform the same action on 12 different Excel workbooks, you can record a macro while you perform the task on the first workbook and then let the macro repeat your action on the other workbooks.
- *Creating a custom command*—If you often issue the same sequence of Excel menu commands, you can save yourself some time by developing a macro that combines these commands into a custom command, which you can execute with a single keystroke or button click.
- *Creating a custom toolbar button*—You can customize the Excel toolbars with your own buttons that execute the macros you write.
- *Creating a custom menu command*—You can customize Excel's menus with your own commands that execute macros you write.
- *Creating a simplified front end*—In almost any office, you can find people who don't really understand how to use computers. Using VBA, you can make it easy for these users to extend their capabilities.
- *Developing new worksheet functions*—Although Excel includes numerous built-in functions (such as **SUM** and **AVERAGE**), you can create custom worksheet functions that can greatly simplify your formulas.
- *Creating complete, macro-driven applications*—If you're willing to spend some time, you can use VBA to create large-scale applications, complete with custom dialog boxes, onscreen help, and lots of other enhancements.

VBA IN A NUTSHELL

LEARNING OUTCOME 3

We will go into much more detail throughout this module; however, here is a brief summary of what VBA is all about:

- You perform actions in VBA by writing (or recording) code in a VBA macro. You view and edit VBA macros using the Visual Basic Editor (VBE).
- A VBA macro consists of Sub procedures. A ***Sub procedure*** is computer code that performs some action on or with objects. The following example shows a simple Sub procedure called *Demo.* This Sub procedure displays the result of 1 plus 1.

```
Sub Demo()
  Sum = 1 + 1
  MsgBox "The answer is " & Sum
End Sub
```

- A VBA macro can also have Function procedures. A ***Function procedure*** is a VBA macro that returns a single value. You can call it from another VBA macro or even use it as a function in a worksheet formula. Here is an example of a Function procedure (named *AddTwo*) that accepts two numbers (called *arguments*) and returns the sum of those values:

```
Function AddTwo(arg1, arg2)
  AddTwo = arg1 + arg2
End Function
```

LEARNING OUTCOME 4

- VBA manipulates objects. An ***object*** in VBA is an item available for you to control in your code. Excel provides more than 100 objects that you can manipulate. Examples of objects include a workbook, a worksheet, a cell range, a chart, and a shape. Figure M.1 provides more information about objects and a few examples.

Figure M.1

VBA Added Terminology and Examples

VBA Function	Explanation	Example
Objects	• Objects are arranged in a hierarchy.	Application.Workbooks ("Book1.xls") Application.Workbooks ("Book1.xls").Worksheets ("Sheet1")
	• Objects can act as *containers* for other objects.	
	• At the top of the object hierarchy is Excel.	
	• Excel itself is an object called *Application,* and it contains other objects such as *Workbook* objects.	
	• The Workbooks collection is contained in the *Application* object.	
	• The *Workbook* object can contain other objects, such as *Worksheet* objects and *Chart* objects.	
Methods	• Objects have methods.	Worksheets("Sheet1").Range ("A1").ClearContents
	• A method is an action Excel performs within an object.	
	• One of the methods for a Range object is *ClearContents,* which clears the contents of the range.	

- You can assign values to variables. A ***variable*** is a place to store a piece of information. You can use variables in your VBA macro to store such things as values, text, or property settings (such as a font color or alignment). To assign the value in cell A1 on Sheet1 to a variable called *Interest,* you could use the following VBA statement:

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

The Visual Basic Editor

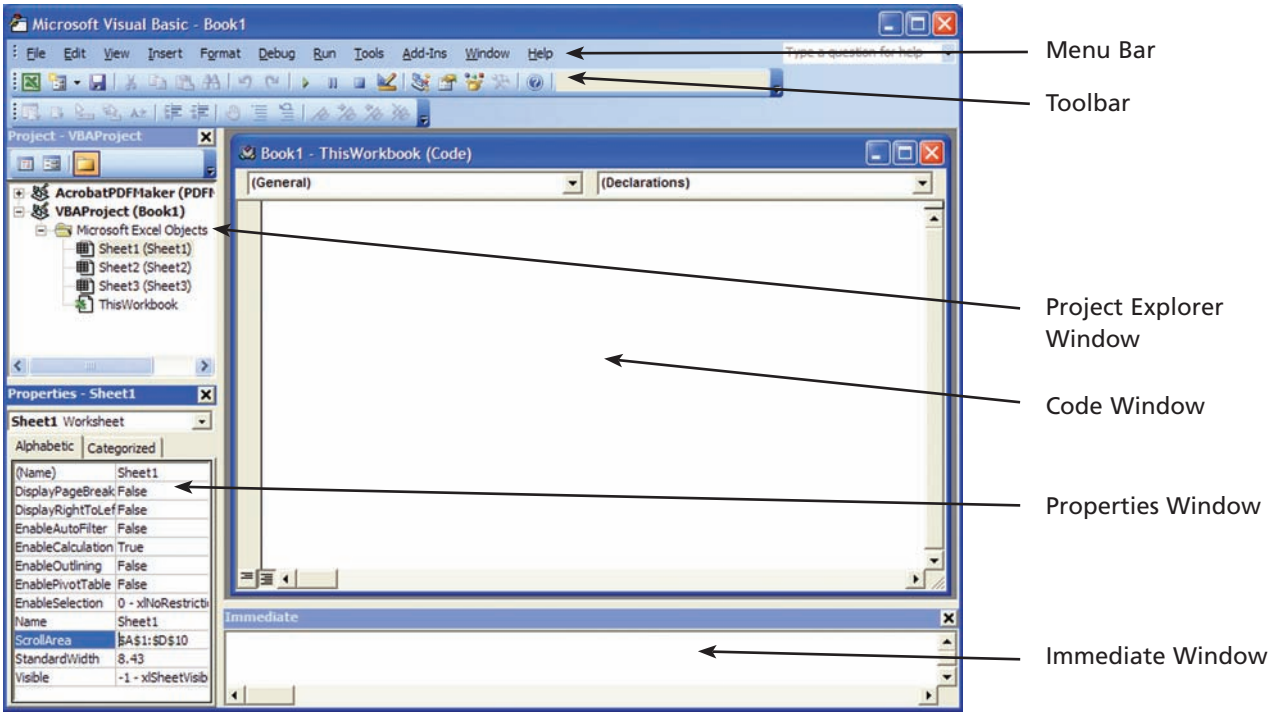
The ***Visual Basic Editor (VBE)*** is a separate application where you write and edit your Visual Basic macros. You can't run the VBE separately; Excel must be running in order for the VBE to operate.

The quickest way to activate the VBE is to press **Alt+F11** when Excel is active. To return to Excel, press **Alt+F11** again. (**Alt+F11** acts as a toggle between the Excel application interface and the VBE.) You can also activate the VBE by using the menus within Excel. To do this, choose **Tools,** then **Macro,** and then choose **Visual Basic Editor.**

THE VBE TOOLS

Figure M.2 shows the VBE program window, with some of the key interface elements identified. Chances are your VBE program window won't look exactly like the window shown in Figure M.2. This window is highly customizable—you can hide, resize, dock, rearrange, and so on in the window. The VBE has even more parts than are shown in Figure M.2, but for the sake of simplicity this module will talk only about what is currently visible.

Figure M.2
VBE Program Window



MENU BAR The VBE menu bar works like every other menu bar you've encountered. It contains commands that you use to do things with the various components in the VBE. You will also find that many of the menu commands have shortcut keys associated with them.

The VBE also features shortcut menus. You can right-click virtually anything in the VBE and get a shortcut menu of common commands.

TOOLBAR The Standard toolbar, which is directly under the menu bar by default (see Figure M.2), is one of four VBE toolbars available. VBE toolbars work just like those in Excel—you can customize them, move them around, display other toolbars, and so on. Choose **View,** then the **Toolbars** command to work with VBE toolbars.

PROJECT EXPLORER WINDOW The Project Explorer window displays a tree diagram that consists of every workbook currently open in Excel (see Figure M.2). If the Project Explorer window is not visible, press **Ctrl+R** or choose **View,** then the **Project Explorer** command. To hide the Project Explorer window, click the **Close** button in the title bar (or right-click anywhere in the Project Explorer window and select **Hide** from the shortcut menu).

CODE WINDOW A Code window contains VBA code. Every object in a project has an associated Code window. To view an object's Code window, double-click the object in the Project Explorer window. For example, to view the Code window for the Sheet1 object, double-click Sheet1 in the Project Explorer window. Unless you've added some VBA code, the Code window will be empty (see Figure M.2). To close the code window, right-click and then select "hide" or click on the close button.

THE PROPERTIES WINDOW The Properties window shows you the properties that can be changed for the currently active object in the Project Explorer window. For example, the *ScrollArea* property in Figure M.2 has been set to A1:D10 to restrict users to that area of the worksheet. Use View/Properties Window or press F4 to show the window. To close the code window, right-click and then select "hide" or click on the close button.

THE IMMEDIATE WINDOW The Immediate window may or may not be visible (see Figure M.2). If it isn't visible, press **Ctrl+G** or choose the **View,** then the **Immediate Window** command. To close the Immediate window, click the **Close** button in the title bar (or right-click anywhere in the Immediate window and select **Hide** from the shortcut menu). The Immediate window is most useful for executing VBA statements directly and for debugging your code.

WORKING WITH THE PROJECT EXPLORER

When you're working in the VBE, each Excel workbook that's open is a project. You can think of a *project* as a collection of objects arranged as an outline. You can expand a project by clicking the plus sign (+) at the left of the project's name in the Project Explorer window. To contract a project click the minus sign (−) to the left of a project's name. Figure M.2 shows a Project Explorer window with three projects listed (Sheet1, Sheet2, and Sheet3).

Every project expands to show at least one *node* called Microsoft Excel Objects. This node expands to show an item for each sheet in the workbook (each sheet is considered an object), and another object called *ThisWorkbook* (which represents the Workbook object as displayed in Figure M.2). If the project has any VBA macros, the project listing also shows a Modules node.

ADDING A NEW VBA MODULE Follow these steps to add a new VBA module to a project:

1. Create a new workbook in Excel.
2. Press **Alt+F11** to activate the VBE.
3. Select the project's name (typically it will be named *ThisWorkbook*) in the Project Explorer window.
4. Choose **Insert** and then **Module** or you can use the shortcut, by using the right mouse click, choosing **Insert,** and then **Module.**

As a note, when you record a macro (which will be discussed shortly), Excel automatically inserts a VBA module (by default it is named *Module1*) to hold the recorded code.

REMOVING A VBA MODULE If you need to remove a VBA module from a project, follow these steps:

1. Select the module's name in the Project Explorer window.
2. Choose **File,** and then **Remove ModuleName** (where **ModuleName** is the name of the module).

You can remove VBA modules, but there is no way to remove the other code modules—those for the *Sheet* objects, or *ThisWorkbook.*

CREATING A MODULE In general, a VBA module can hold several types of code:

- *Sub procedures*—A set of programming instructions that performs some action (we described this earlier and will go into more detail shortly).
- *Function procedures*—A set of programming instructions that returns a single value. This is similar in concept to a worksheet function such as **SUM** described earlier; more detail will be provided shortly.
- *Declarations*—One or more information statements that you provide to VBA. For example, you can declare the data type for variables you plan to use, or set some other module options. Although declarations are mentioned (as they relate to modules), it is beyond the scope of this module to go into any more detail.

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

VBA MODULE CODE

Before you can do anything meaningful, you must have some VBA code in the VBA module. You can get VBA code into a VBA macro in two ways:

1. Entering the code directly by typing it.
2. Using the Excel macro recorder to record your actions and convert them to VBA code.

ENTERING CODE DIRECTLY You can type code directly into the module. You can select, copy, cut, paste, and do other things to the text. When you are entering your code directly, you use the **Tab** key to indent some of the lines to make your code easier to read. This isn't necessary but it's a good habit to acquire. A single line of VBA code can become very long. Therefore, you may want to use the line continuation character to

break up lengthy lines of code. To continue a single line of code from one line to the next, end the first line with a space followed by an underscore (_), then continue the statement on the next line. Here's an example of a single line of code split into three lines:

```
Selection.Sort Key1:=Range("A1"), _
  Order1:=xlAscending, Header:=xlGuess, _
  Orientation:=xlTopToBottom
```

This statement would perform exactly the same way if it were entered in a single line (with no line continuation characters). Notice that the second and third lines of this statement are indented. Indenting makes it clear that these lines are not separate statements.

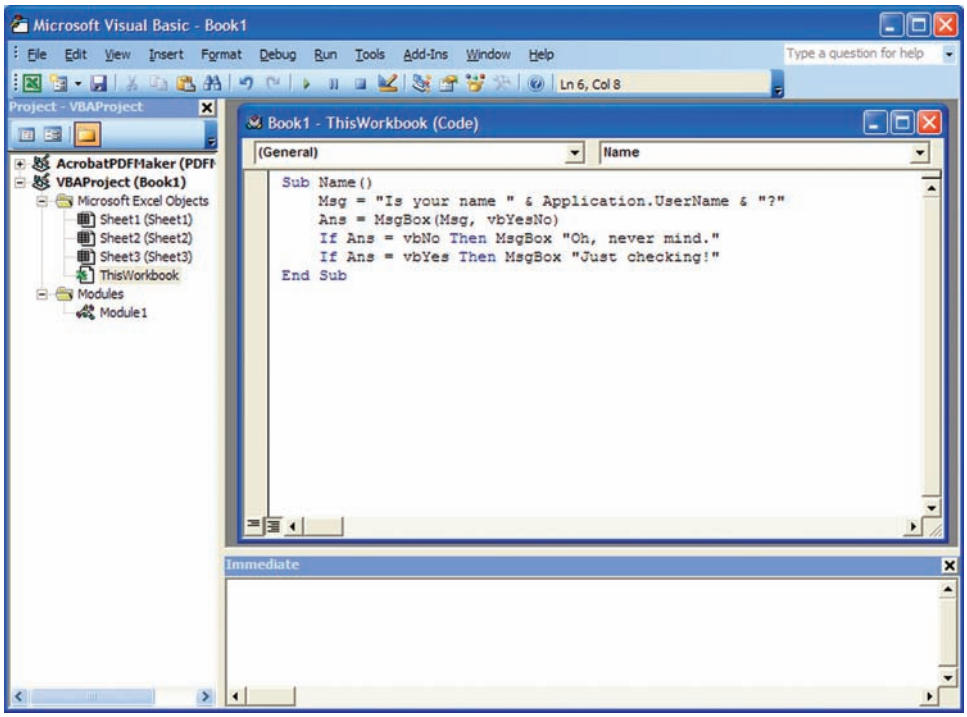
The Visual Basic Editor (VBE) has multiple levels of undo and redo. Therefore, if you deleted a statement that you shouldn't have, use the **Undo** button on the toolbar until the statement comes back. After undoing, you can use the **Redo** button to perform the changes you've undone.

OK, it's time to enter some real code. Try the following steps:

1. Create a new workbook in Excel.
2. Press **Alt+F11** to activate the VBE.
3. Click the new workbook's name in the Project Explorer window.
4. Choose **Insert,** then **Module** to insert a VBA module into the project.
5. Type the following code into the Code window (see Figure M.3):

```
Sub NameDemo()
  Msg = "Is your name" & Application.UserName & "?"
  Ans = MsgBox(Msg, vbYesNo)
  If Ans = vbNo Then MsgBox "Oh, never mind."
  If Ans = vbYes Then MsgBox "Just checking!"
End Sub
```

Figure M.3
The Code Window



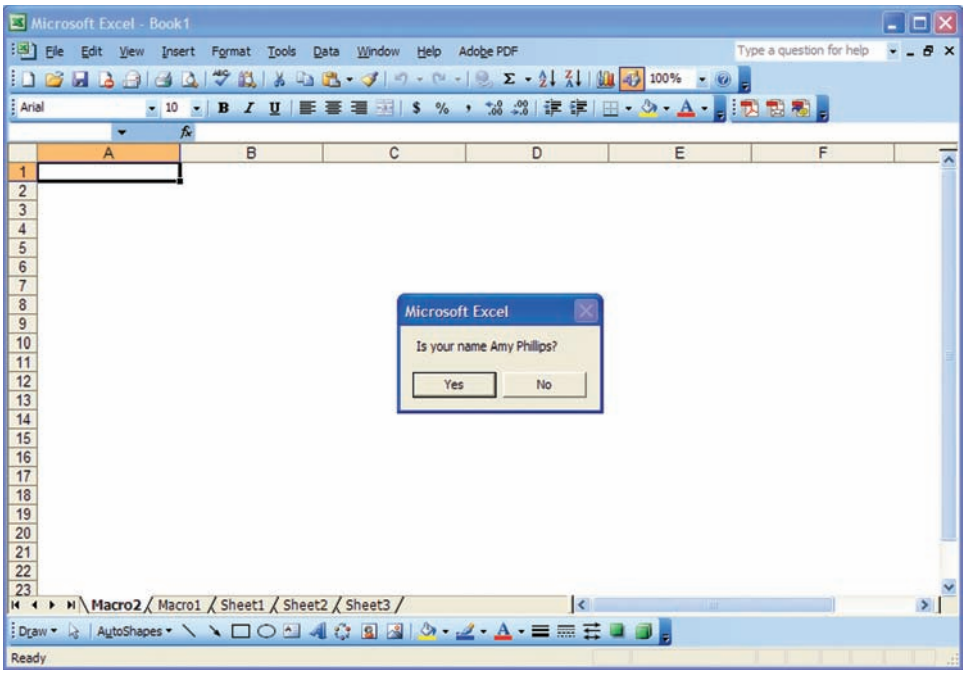


Figure M.4
Run Code Execution

6. Make sure the cursor is located anywhere within the text you typed.
7. Press **F5** to execute the procedure (**F5** is a shortcut for the **Run, Run Sub/ UserForm** command).
8. If you entered the code correctly, Excel executes the procedure and you can respond to the dialog box shown in Figure M.4.

When you enter the code listed in Step 5, you might notice that the VBE makes some adjustments to the text you enter. For example, after you type the **Sub** statement, the VBE automatically inserts the **End Sub** statement. And if you omit the space before or after an equal sign, the VBE inserts the space for you. Also, the VBE changes the color and capitalization of some text. This is all perfectly normal, so don't *Undo* any of this.

If you followed the previous steps, you've just written a VBA Sub procedure, or rather a *macro.* This macro, albeit simple, uses the following concepts:

- Defining a Sub procedure (the first line).
- Assigning values to variables (**Msg** and **Ans**).
- *Concatenating* (i.e., joining) a string (using the **&** operator).
- Using a built-in VBA function (**MsgBox**).
- Using built-in VBA constants (**vbYesNo, vbNo,** and **vbYes**).
- Using an If-Then construct (twice).
- Ending a Sub procedure (the last line).

The concepts mentioned in the bullet list above will be described in detail later in this module.

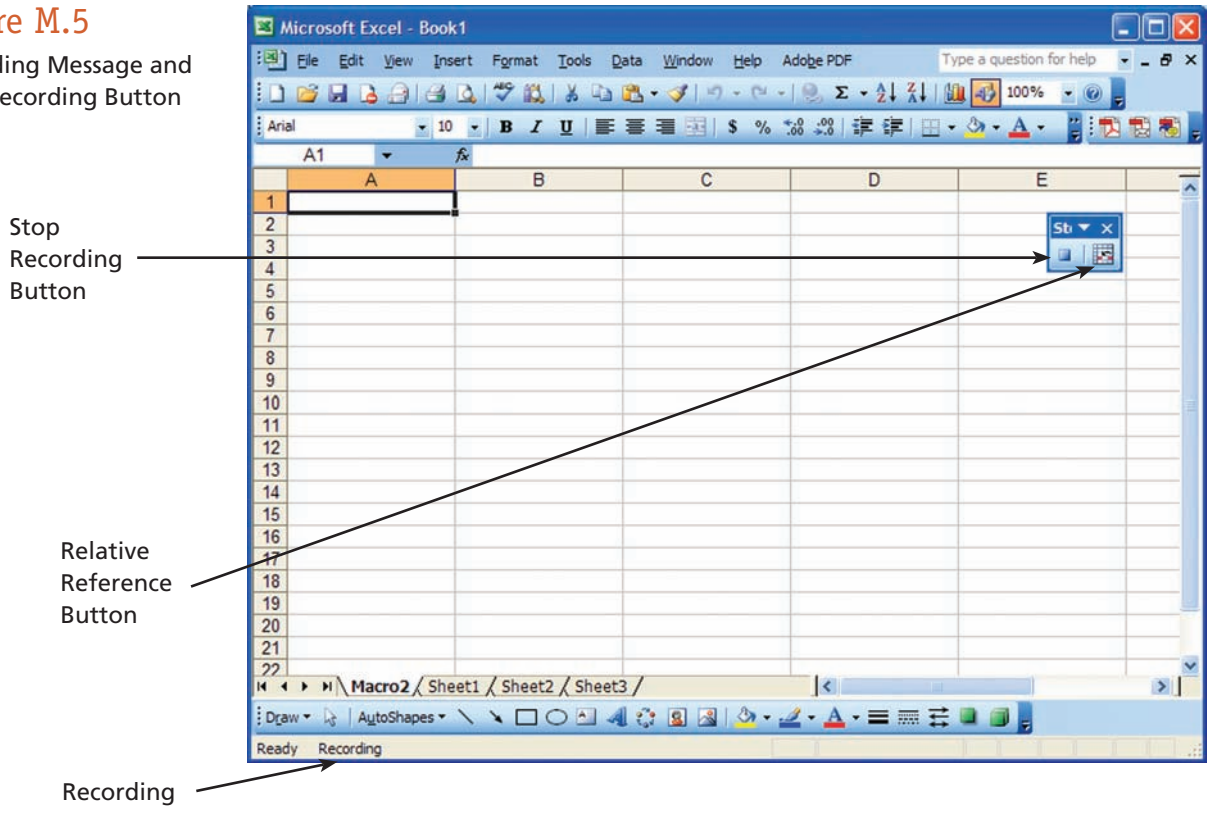
This is a good time to save what you have been working on. When you save an Excel workbook, any macro that you have created automatically gets saved within the workbook, so there is nothing different or new you need to do other than clicking on **File, Save.**

USING THE MACRO RECORDER Another way you can get code into a VBA macro is by recording your actions using the Excel macro recorder. There is no way you can record the *NameDemo* procedure shown in the preceding section since you can record only things that you can do directly in Excel. Displaying a message box is not in Excel's normal repertoire. The macro recorder is useful, but in many cases you'll probably have to enter at least some code manually.

Here's a step-by-step example that shows you how to record a macro that turns off the cell gridlines in a worksheet. Follow these steps:

1. Activate a worksheet in the workbook.
2. Choose **Tools,** then **Macro,** and then **Record New Macro.**
3. Excel displays its Record Macro dialog box.
4. Click **OK** to accept the defaults.
5. Excel automatically inserts a new VBA module into the project that corresponds to the active workbook.
6. From this point on, Excel converts your actions into VBA code—while recording, Excel displays the word *Recording* in the status bar (see Figure M.5).
7. Excel displays a miniature floating toolbar that contains two toolbar buttons: *Stop Recording* and *Relative Reference* (see Figure M.5).
8. Choose **Tools,** then **Options.** Excel displays its Options dialog box.
9. Click the **View** tab.
10. Remove the check mark from the **Gridlines** option. If the worksheet you're using has no gridlines, put a check mark next to the **Gridlines** option.
11. Click **OK** to close the dialog box.
12. Click the **Stop Recording** button on the miniature toolbar—Excel stops recording your actions.

Figure M.5
Recording Message and Stop Recording Button



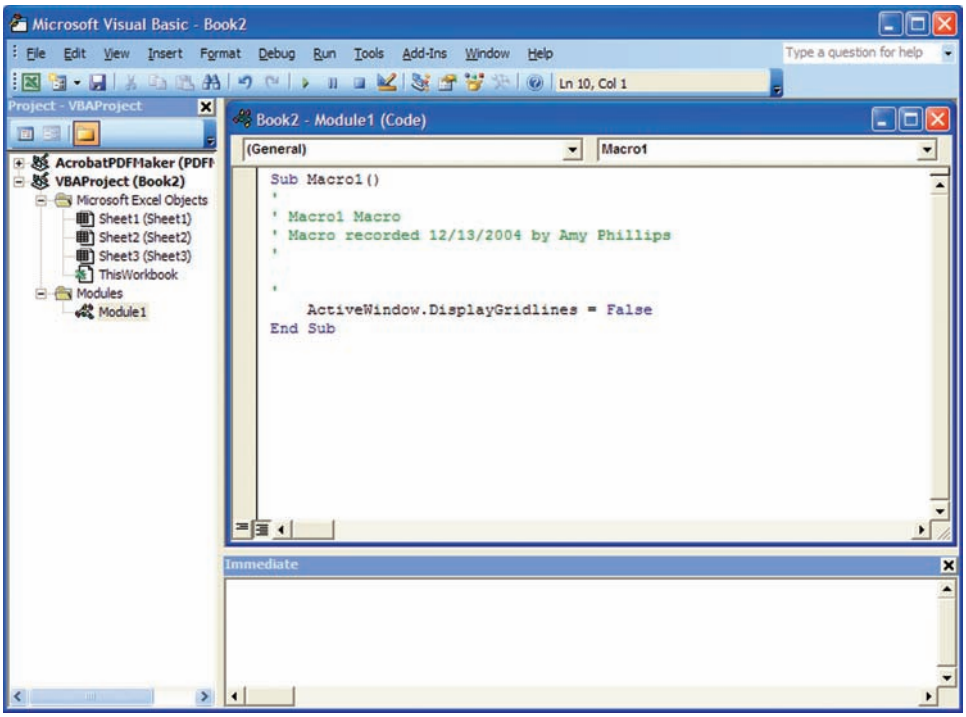


Figure M.6
Module1 Macro Code

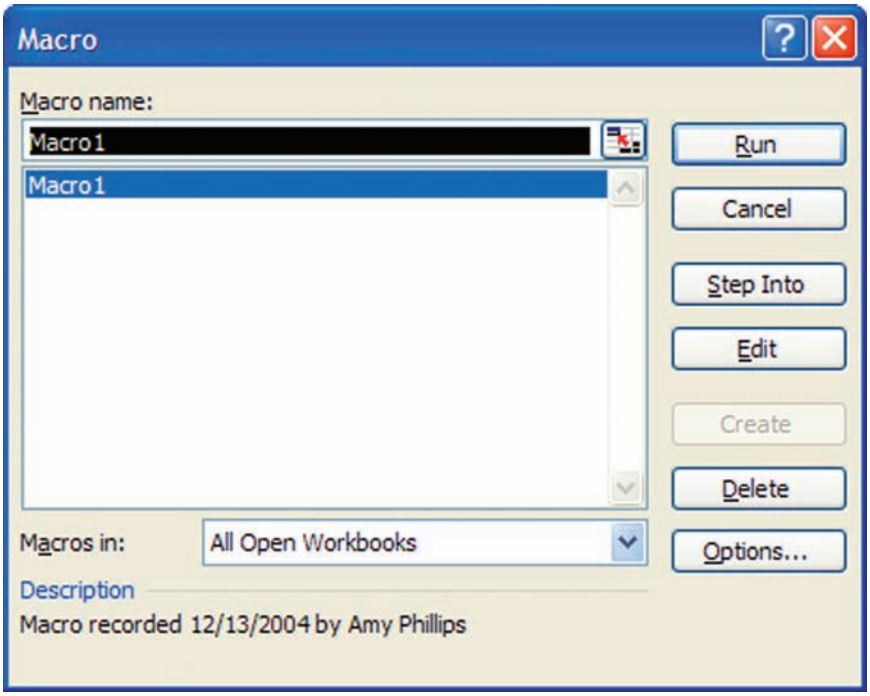
To view this newly recorded macro, press **Alt+F11** to activate the VBE. Locate the workbook's name in the Project Explorer window. You'll see that the project has a new module listed. The name of the module depends on whether you had any other modules in the workbook when you started recording the macro. If you didn't, the module will be named *Module1*. You can double-click the module to open the Code window. Figure M.6 displays the code for this example.

Now let's give this macro a try:

1. Activate a worksheet that has gridlines displayed.
2. Choose **Tools,** then **Macro,** and then choose **Macros,** or press **Alt+F8.**
3. Select **Macro1** (see Figure M.7).
4. Click the **Run** button.
5. Excel executes the macro, and the gridlines disappear.

You can execute any number of commands and perform a variety of actions while the macro recorder is running. Excel translates your mouse actions and keystrokes to VBA code.

Figure M.7
Macro Dialog Box



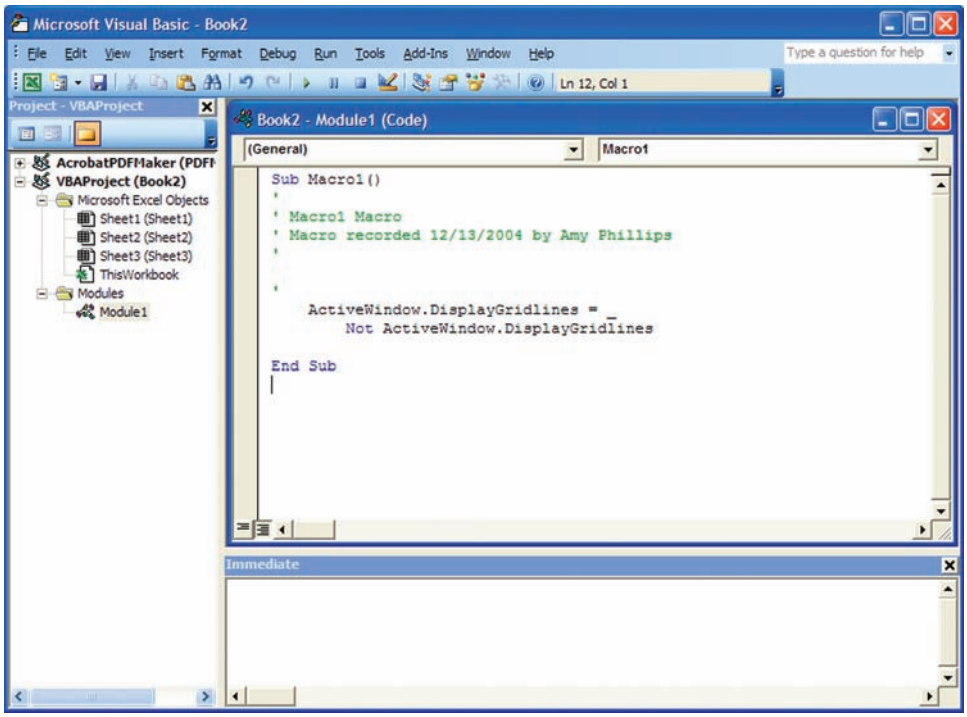


Figure M.8
Display Gridlines Macro

The preceding macro is a great demonstration of the macro recorder, but really isn't all that practical. To make it truly functional, activate the module, and change the statement to this (refer to Figure M.8):

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

This modification makes the macro serve as a toggle. If gridlines are displayed, the macro turns them off. If gridlines are not displayed, the macro turns them on. Now run the macro within the spreadsheet itself by closing the VBA window and pressing **Alt+F8,** then the **Run** button.

Another way to execute a macro is to press its shortcut key. Before you can use this method, you must assign a shortcut key to the macro.

You have the opportunity to assign a shortcut key in the Record Macro dialog box when you begin recording a macro. If you create the procedure without using the macro recorder, you can assign a shortcut key (or change an existing shortcut key) using the following procedure:

1. Choose **Tools,** then **Macro,** and then **Macros.**
2. Select the **Macro1** Sub procedure name (that was created in the previous step) from the list box.
3. Click the **Options** button.
4. Click the **Shortcut Key** option and enter a letter in the box labeled **Ctrl +.** The letter you enter corresponds to the key combination you want to use for executing the macro.
5. Click **OK** to close the Macro Options dialog box.

After you've assigned a shortcut key, you can press that key combination to execute the macro.

VBA Building Blocks

VBA is easy to learn but can be considered a serious programming language. It can be used to perform complex tasks such as automatically getting up-to-date financial information from the Internet and calculating option prices, and can be used in scientific applications.

There are many ways to write a macro using Excel VBA. You can write or record macros using modules or procedures, or you can develop user-defined functions. These are some of the simple-to-understand building blocks to learn within the VBA structure.

CODE MODULES

All macros reside in code modules like the one on the right of the VBE window (refer back to Figure M.2). There are two types of code modules—(1) standard modules and (2) class modules. The one you see in Figure M.2 is a standard module. You can use class modules to create your own objects (which is beyond our scope here).

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

PROCEDURES

In VBA, macros are referred to as procedures. There are two types of procedures—(1) Sub procedures and (2) Function procedures. The macro recorder can produce only Sub procedures.

SUB PROCEDURES Sub procedures (sometimes referred to as *subroutines*) start with the keyword *Sub* followed by the name of the procedure and opening and closing parentheses. The end of a Sub procedure is marked by the keywords *End Sub.* By convention, the code within the Sub procedure is indented to make it stand out from the start and end of the Sub procedure, so that the code is easier to read. Here is an example of a Sub procedure:

```
Sub MonthNames()
'
' MonthNames Macro
' Macro recorded 12/20/2004 by Amy Phillips
'
    Range("B1").Select
    ActiveCell.FormulaR1C1 = "Jan"
    Range("C1").Select
    ActiveCell.FormulaR1C1 = "Feb"
    Range("D1").Select
    ActiveCell.FormulaR1C1 = "Mar"
    Range("E1").Select
    ActiveCell.FormulaR1C1 = "Apr"
    Range("F1").Select
    ActiveCell.FormulaR1C1 = "May"
    Range("G1").Select
    ActiveCell.FormulaR1C1 = "Jun"
    Range(Selection, Selection.End(xlToLeft)).Select
    Selection.Font.Italic = True
    Selection.Font.Bold = True
    Range("A2").Select
End Sub
```

If you look at the code in *MonthNames* Sub procedure, you will see that cells are being selected and then the month names are assigned to the active cell formula. You can edit some parts of the code, so if you had spelled a month abbreviation incorrectly, you could fix it; or you could identify and remove the line that sets the font to bold; or you could select and delete an entire macro.

As a note, any lines starting with a single quote are comment lines, which are ignored by VBA (which will be discussed in more detail in the next section). They are added to provide documentation.

FUNCTION PROCEDURES Excel has hundreds of built-in worksheet Function procedures (or simply referred to as functions) that you can use in cell formulas. You can select an empty worksheet cell and choose the **Insert,** and then the **Function** command to see a list of those functions. Among the most frequently used functions are **SUM, IF,** and **VLOOKUP.** If the function you need is not already in Excel, you can write your own *user defined function* (or UDF) using VBA.

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

Unlike manual operations, UDFs cannot be recorded. You have to write them from scratch using a standard module in the VBE. If necessary, you can insert a standard module by right-clicking in the Project Explorer window and choosing **Insert,** then **Module.** Consider the following:

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

This function illustrates an important concept about functions: how to return the value that makes functions so important. Notice that the single line of code that makes up this Function procedure is a formula. Here, you create a function called *CentigradeToFahrenheit()* that converts degrees Centigrade to degrees Fahrenheit.

Connect to the Web site that supports this text (www.mhhe.com/haag select *XLM/M*) and download the file called **XLMM_UDF.xls.** In the worksheet, column A contains degrees Centigrade, and column B uses the *CentigradeToFahrenheit()* Function procedure to calculate the corresponding temperature in degrees Fahrenheit. You can see the formula in cell B2 by looking at the Formula bar.

Remember that the key difference between a Sub procedure and a Function procedure is that a Function procedure returns a value. *CentigradeToFahrenheit()* calculates a numeric value, which is returned to the worksheet cell where *CentigradeToFahrenheit()* is used.

You need to open the VBE in order to review the Function procedure macro:

1. Choose **Tools** on the Menu bar.
2. Select **Macro,** then **Visual Basic Editor,** or **Alt+F11.**

Take a look at the Function procedure macro (double-click on *Module1*). Normally Function procedures have one or more input parameters. *CentigradeToFahrenheit()* has one input parameter called *Centigrade,* which is used to calculate the return value. When you enter the formula, **=CentigradeToFahrenheit(A2),** the value in cell A2 is passed to *CentigradeToFahrenheit()* through the input parameter *Centigrade.* For example, if the value of *Centigrade* is 0 (zero), *CentigradeToFahrenheit()* sets its own name equal to the calculated result, which is 32. The result is passed back to cell B2. The same process occurs in each cell that contains a reference to *CentigradeToFahrenheit().*

Elements of VBA Programming

LEARNING OUTCOME 5

VBA uses many elements common to all programming languages, such as: comments, variables, constants, data types, and others. If you've programmed using other computer languages, some of this material will be familiar. If this is your first experience programming, it should be an enjoyable exercise.

COMMENTS

A *comment* is the simplest type of VBA statement. Because VBA ignores these statements, they can consist of anything you want. You can insert a comment to remind yourself why you did something or to clarify a piece of code you wrote.

You begin a comment with a single quote (‘). VBA ignores any text that follows a single quote in a line of code. You can use a complete line for your comment or insert your comment at the end of a line of code. The following example shows a VBA procedure with three comments:

```
Sub CommentsExample()
  ' This procedure is a demonstration
  x = 0 ' x represents zero
  ' The next line of code will display the result
  MsgBox x
End Sub
```

VARIABLES AND CONSTANTS

VBA’s main purpose is to manipulate data. VBA stores the data in your computer’s memory, where some data, such as worksheet ranges, reside in objects and other data are stored in variables or constants that you create.

VARIABLES As defined earlier, a *variable* is the name of a storage location. You have lots of flexibility in naming your variables, so make the variable names as descriptive as possible. You assign a value to a variable using the equal sign operator. Here are a few examples that use variable names. The variable names are on the left side of the equal signs (note that the last example uses two variables):

```
x = 1
InterestRate = 0.075
LoanPayoffAmount = 243089
DataEntered = False
x = x + 1
UserName = "Amy Phillips"
DateStarted = #12/20/2004#
MyNum = YourNum * 1.25
```

VBA enforces a few rules regarding variable names:

- You can use letters, numbers, and some punctuation characters, but the first character must be a letter.
- You cannot use any spaces or periods in a variable name.
- VBA does not distinguish between uppercase and lowercase letters.
- You cannot use the *#, $, %, &, ‘,* or *!* characters in a variable name.
- Variable names can be no longer than 254 characters (although it is not recommended to use more than 20 characters because they become hard to read).

To make variable names more readable, programmers often use mixed case (for example, InterestRate) or the underscore character (Interest_Rate). VBA has many reserved words that you can’t use for variable names or procedure names, for example:

- Built-in VBA function names such as *Ucase* and *Sqr.*
- VBA language words such as *Sub, With,* and *For.*

If you attempt to use one of these names as a variable, you may get a compile error (i.e., your macro won’t run). So, if an assignment statement produces an error message, double check and make sure that the variable name isn’t a reserved word.

CONSTANTS A variable's value may (and usually does) change while your procedure is executing. Sometimes, you need to refer to a value or string that never changes, in other words, a *constant*. A ***constant*** is a named element whose value doesn't change. Here are a few examples that declare constants by using the *Const* statement:

```
Const NumQuarters As Integer = 4
Const Rate = .0725, Period = 12
Const ModName As String = "Budget Macros"
```

Using constants in place of hard-coded values or strings (i.e., something other than a value) is an excellent programming practice. For example, if your procedure needs to refer to a specific value (such as an interest rate) several times, it's better to declare the value as a constant and refer to its name rather than the value. This makes your code more readable and easier to change; should the need for changes arise, you have to change only one statement rather than several.

DATA TYPES Data types are the manner in which data types are stored in memory—for example, as integers, real numbers, or strings. VBA has a variety of built-in data types. Figure M.9 lists the most common types of data that VBA can handle.

LEARNING OUTCOME 6

STRINGS Excel and VBA can work with both numbers and text. Text is often referred to as a *string*. There are two types of strings in VBA:

1. *Fixed-length strings* are declared with a specified number of characters (the maximum length is about 65,526 characters).
2. *Variable-length strings* theoretically can hold as many as 2 billion characters.

Data Type	Bytes Used	Range of Values
Boolean	2	True or False
Integer	2	−32,768 to 32,767
Long	4	−2,147,483 to 2,147,483,647
Single	4	−3.402823E38 to 1.401298E45
Double (negative)	8	−1.79769313486232E308 to −4.94065645841247E-324
Double (positive)	8	4.94065645841247E-324 to 1.79769313486232E308
Currency	8	−922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date	8	1/1/100 to 12/31/9999
String	1 per char	Varies
Object	4	Any defined object
Variant	Varies	Any data type
User defined	Varies	Varies

Figure M.9
VBA Built-in Data Types

So far you have been creating variables simply by using them. This is referred to as implicit variable declaration. Most computer languages require us to employ *explicit variable declaration.* This means that you must define the names of all the variables you are going to use, before you use them in your code. VBA allows both types of declaration. If you want to declare a variable explicitly, you do so using a **Dim** statement. When declaring a string variable with a **Dim** statement, you can specify the maximum length if you know it (it's a fixed-length string) or let VBA handle it dynamically (it's a variable-length string). The following example declares the *MyString* variable as a string with a maximum length of 50 characters (use an asterisk to specify the number of characters, up to the 65,526 character limit). *YourString* is also declared as a string but its length is unspecified (which is typically what is recommended that you use):

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

DATES Although you can use a string variable to store dates, it is recommended that you use the Date data type. If you do, you will be able to perform calculations with the dates. For example, you might need to calculate the number of days between two dates. This would be impossible if you used strings to hold your dates.

A variable defined as a date can hold dates ranging from January 1, 0100, to December 31, 9999. That's a span of nearly 10,000 years. You can also use the date data type to work with time data.

Here are a few examples that declare variables and constants as a Date data type (note that in VBA, dates and times are placed between two hash marks, i.e., the # symbols):

```
Dim Today As Date
Dim StartTime As Date
Const FirstDay As Date = #1/1/2005#
Const Noon = #12:00:00#
```

Date variables display dates according to your system's short date format, and display times according to your system's time format (either 12- or 24-hour). Therefore, the VBA-displayed date or time format may vary, depending on the settings for the system on which the application is running.

ASSIGNMENT STATEMENTS

An *assignment statement* is a VBA statement that assigns the result of an expression to a variable or an object. Excel's Help system defines the term *expression* as:

a combination of keywords, operators, variables, and constants that yields a string, number, or object. An expression can be used to perform a calculation, manipulate characters, or test data.

Much of your work in VBA involves developing (and sometimes debugging) expressions. If you know how to create formulas in Excel, you'll have no trouble creating expressions. With a worksheet formula, Excel displays the result in a cell. A VBA expression, on the other hand, can be assigned to a variable. Here are a few assignment statement examples (the expressions are to the right of the equal sign):

```
x = 1
x = x + 1
x = (y * 2) / (z * 2)
HouseCost = 375000
FileOpen = True
Range("TheYear").Value = 2005
```

OPERATORS

Operators play a major role in VBA. Besides the equal sign operator (=), VBA provides several other operators, as presented in Figure M.10.

The precedence order for operators in VBA is exactly the same as in Excel formulas. Exponentiation has the highest precedence. Multiplication and division come next, followed by addition and subtraction. You can use parentheses to change the natural precedence order, making whatever's sandwiched in parentheses come before any operator.

Operator	Operator Symbol
Addition	+
Multiplication	*
Division	/
Subtraction	-
Exponentiation	^
String concatenation	&
Integer division (the result is always an integer)	\
Modulo arithmetic (returns the remainder of a division operation)	Mod
Logical Operator	**What It Does**
Not	Negation on an expression
And	Conjunction on two expressions
Or	Disjunction on two expressions
XoR	Exclusion on two expressions
Eqv	Equivalence on two expressions
Imp	Implication on two expressions

Figure M.10
VBA Operators

LEARNING OUTCOME 7

Decisions, Decisions, Decisions

Some VBA procedures start at the code's beginning and progress line by line to the end, never deviating from this top-to-bottom flow. Macros that you record always work like this. In many cases, however, you need to control the flow of your code by skipping over some statements, executing some statements multiple times, and testing conditions to determine what the procedure does next.

VBA is indeed a structured language. It offers standard structured decision constructs such as *If-Then* and *Select Case* structures, and *For-Next, Do-Until,* and *Do-While* loops.

THE IF-THEN STRUCTURE

The *If-Then* statement is VBA's most important control structure. The *If-Then* structure has this basic syntax:

```
If condition Then statements [Else elsestatements]
```

Use the *If-Then* structure when you want to execute one or more statements conditionally. The optional *Else* clause, if included, lets you execute one or more statements if the condition you're testing is not true. When a condition is true, VBA executes the conditional statements and the *If-Then* ends.

OK, let's apply this concept to a spreadsheet scenario. If you had a worksheet that needed to calculate a discount rate based on quantity, you would want to use an *If-Then* structure to help us out. Here's an example that does just that. This procedure uses the value from cell A2 (i.e., **Cells(2, 1).Value**), assigns it to the variable *Quantity,* and then displays the appropriate discount in cell B2 (i.e., **Cells(2, 2).Value**), based on the quantity the user enters:

```
Sub ShowDiscount()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = Cells(2, 1).Value
  If Quantity > 0 Then Discount = 0.1
  If Quantity >= 25 Then Discount = 0.15
  If Quantity >= 50 Then Discount = 0.2
  If Quantity >= 75 Then Discount = 0.25
  Cells(2, 2).Value = Discount
End Sub
```

Notice that each *If-Then* statement in this Sub procedure is executed and the value for *Discount* can change as the statements are executed. However, the procedure ultimately displays the correct value for *Discount.*

The following procedure performs the same tasks by using the alternative *ElseIf* syntax. In this case, the procedure ends immediately after executing the statements for a true condition:

```
Sub ShowDiscount2()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = Cells(2, 1).Value
  If Quantity > 0 And Quantity < 25 Then
    Discount = 0.1
  ElseIf Quantity >= 25 And Quantity < 50 Then
    Discount = 0.15
  ElseIf Quantity >= 50 And Quantity < 75 Then
    Discount = 0.2
  ElseIf Quantity >= 75 Then Discount = 0.25
  End If
  Cells(2, 2).Value = Discount
End Sub
```

THE SELECT CASE STRUCTURE

The *Select Case* structure is useful for decisions involving three or more options (although it also works with two options, providing an alternative to the *If-Then* structure). The syntax for the *Select Case* structure is:

```
Select Case TestExpression
[Case expressionlist-n
  [statements-n]] . . .
End Select
```

The following example shows how to use the *Select Case* structure (this also shows another way to code the examples presented above):

```
Sub ShowDiscount3()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = Cells(2, 1).Value
  Select Case Quantity
    Case 0 To 24
      Discount = 0.1
    Case 25 To 49
      Discount = 0.15
    Case 50 To 74
      Discount = 0.2
    Case Is >= 75
      Discount = 0.25
  End Select
  Cells(2, 2).Value = Discount
End Sub
```

In this example, the *Quantity* variable is being evaluated. The Sub procedure is checking for four different cases (0 to 24, 25 to 49, 50 to 74, and 75 or greater). Any number of statements can follow each *Case* statement, and they all are executed if the case is true.

When VBA executes a *Select Case* structure, the structure is exited as soon as VBA finds a true case.

LOOPING

The term ***looping*** refers to repeating a block of statements or code numerous times. You may know how many times your macro needs to loop, or variables used in your programs may determine this. There are several looping statements to choose from: the *For-Next* loop, the *Do-While* loop, and the *Do-Until* loop.

FOR-NEXT LOOPS The simplest type of loop is a *For-Next* loop. Here's the syntax for this structure:

```
For Counter = Start To End [Step n]
  [statements]
Next [Counter]
```

The looping is controlled by a counter variable, which starts at one value and stops at another value. The statements between the *For* statement and the *Next* statement are the statements that get repeated in the loop.

The following example shows a *For-Next* loop that doesn't use the optional *Step* value. This Sub procedure loops 100 times and uses the VBA **Rnd** function to enter a random number into 100 cells:

```
Sub FillRange()
  Dim Count As Integer
  For Count = 1 To 100
    ActiveCell.Offset(Count − 1, 0) = Rnd
  Next Count
End Sub
```

In this example, *Count* (the loop counter variable) starts with a value of 1 and increases by 1 each time through the loop. Because you didn't specify a Step value, VBA uses the default value (which is 1). The *Offset* method uses the value of *Count* as an argument. The first time through the loop, the procedure enters a number into the active cell offset by zero rows. The second time through (Count = 2), the procedure enters a number into the active cell offset by one row (Count −1), and so on.

***DO-WHILE* LOOPS** VBA supports another type of looping structure known as a *Do-While* loop. Unlike a *For-Next* loop, a *Do-While* loop continues until a specified condition is met. Here's the *Do-While* loop syntax:

```
Do [While condition]
  [statements]
Loop
```

The following example uses a *Do-While* loop within a Sub procedure. This procedure uses the active cell as a starting point and then travels down the column, multiplying each cell's value by 2. The loop continues until the procedure encounters an empty cell.

```
Sub DoWhileExample()
  Do While ActiveCell.Value <> Empty
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(1, 0).Select
  Loop
End Sub
```

***DO-UNTIL* LOOP** The *Do-Until* loop structure is similar to the *Do-While* structure. The two structures differ in their handling of the tested condition. A macro continues to execute a *Do-While* loop while the condition remains true. In a *Do-Until* loop, the macro executes the loop until the condition is true. Here's the *Do-Until* syntax:

```
Do [Until condition]
  [statements]
Loop
```

The following example is the same one presented for the *Do-While* loop but recoded to use a *Do-Until* loop:

```
Sub DoUntilExample()
  Do Until IsEmpty (ActiveCell.Value)
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(l, 0).Select
  Loop
End Sub
```

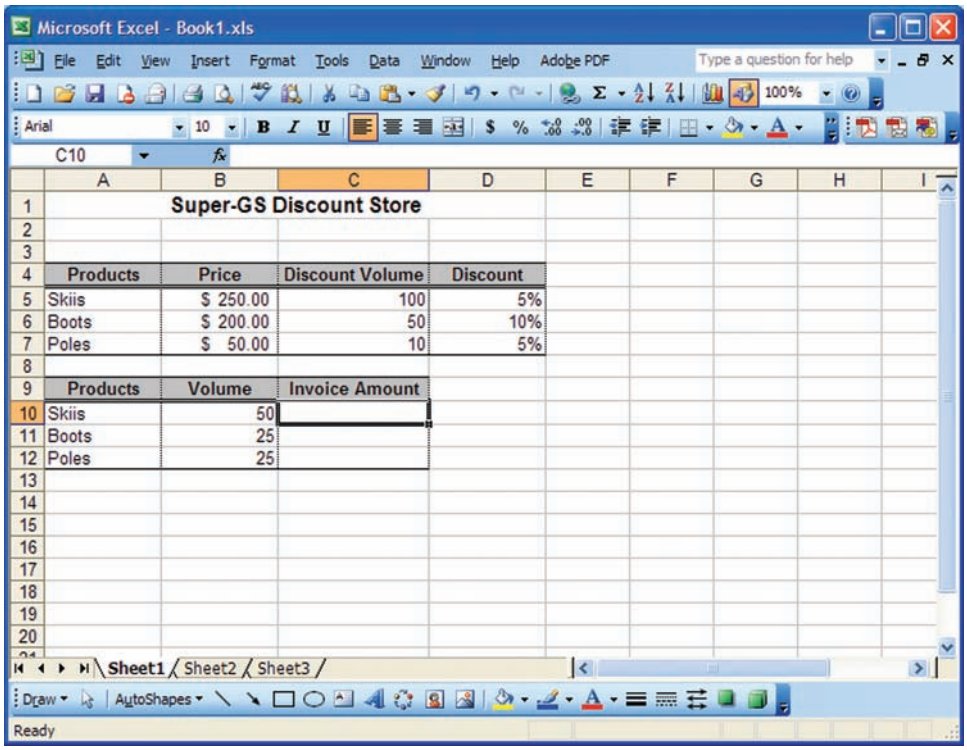
Wrap It Up

Figure M.11 displays a simple worksheet created to calculate the invoice amount of three products based on several parameters. The lookup table in cells A5:D7 gives the price of each product, the discount sales volume (above which a discount will be applied), and the percent discount for units above the discount volume. Using normal spreadsheet formulas, you would have to set up three lookup formulas together with some logical tests to calculate the invoice amount.

Let's take what you have learned in the previous sections and put some of these concepts to work. You can follow the steps below to create this exercise from scratch or you can download the solution and follow along. To download the solution, connect to the Web site that supports this text (www.mhhe.com/haag select **XLM/M**) and select the file called **XLMM_DiscountSales.xls.**

1. Open a new workbook.
2. Create a worksheet with the same structure and data as you see in Figure M.11 (don't worry about the aesthetics).
3. Press **Alt+F11** to activate the VBE.
4. Click the new workbook's name (it should be named *ThisWorkbook*) in the Project Explorer window.

Figure M.11
VBA Example



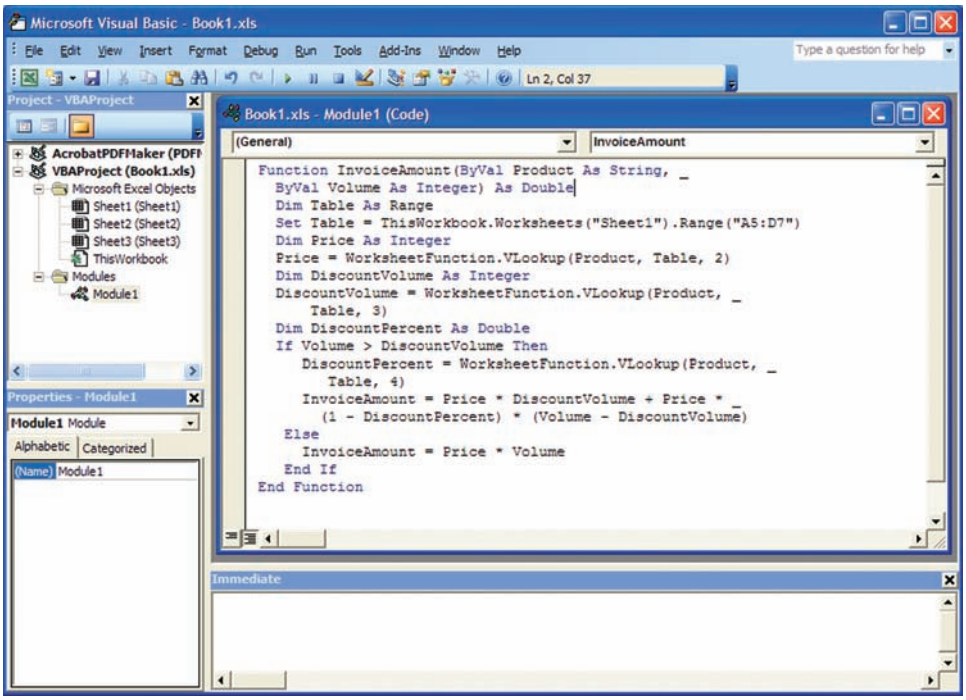


Figure M.12
VBA Example Code

5. Choose **Insert,** then **Module** to insert a VBA module into the project.
6. Type the following code into the Code window (refer to Figure M.12):

```
Function InvoiceAmount(ByVal Product As String, _
    ByVal Volume As Integer) As Double
  Dim Table As Range
  Set Table = ThisWorkbook.Worksheets("Sheet1").Range("A5:D7")
  Dim Price As Integer
  Price = WorksheetFunction.VLookup(Product, Table, 2)
  Dim DiscountVolume As Integer
  DiscountVolume = WorksheetFunction.VLookup(Product, _
    Table, 3)
  Dim DiscountPercent As Double
  If Volume > DiscountVolume Then
    DiscountPercent = WorksheetFunction.VLookup(Product, _
      Table, 4)
    InvoiceAmount = Price * DiscountVolume + Price * _
      (1 – DiscountPercent) * (Volume – DiscountVolume)
  Else
    InvoiceAmount = Price * Volume
  End If
End Function
```

Before you continue, let's review what all this code really means. The *InvoiceAmount()* function has three input parameters: *Product* is the name of the product; *Volume* is the number of units sold, and *Table* is the lookup table. The formula in cell C10 defines the values to be used for each input parameter.

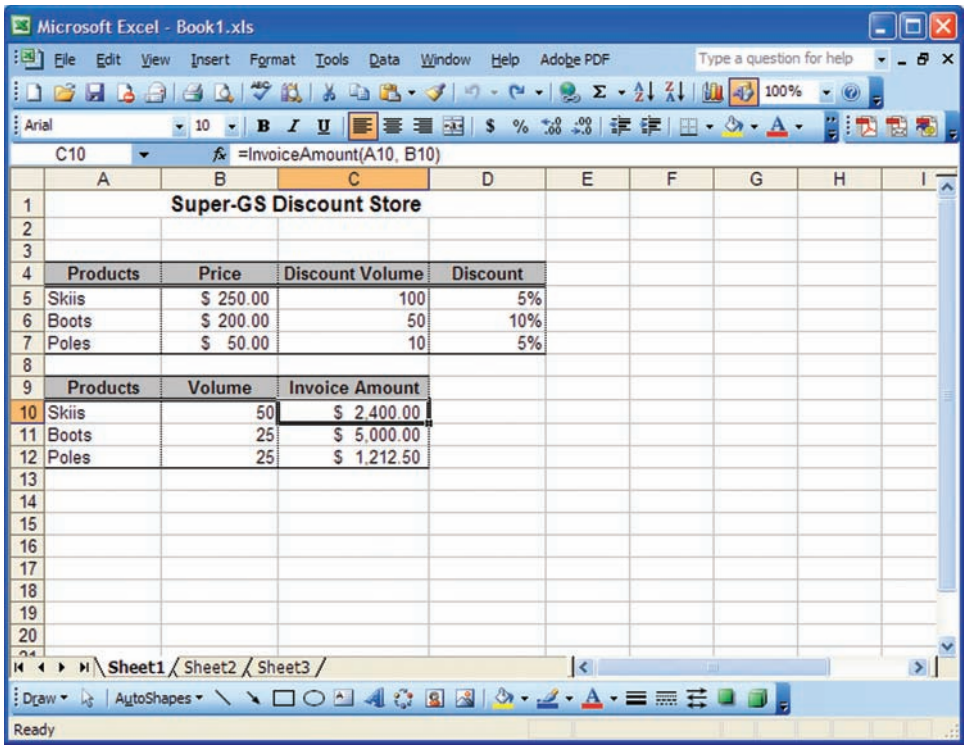
The range for the table is absolute so that copying the formula below cell C7 refers to the same range. The first calculation in the function uses the **VLookup** function to find the product in the lookup table and return the corresponding

value from the second column of the lookup table, which it assigns to the variable *Price.* In the next line of the function, the discount volume is found in the lookup table and assigned to the variable *DiscountVolume.* The *If* test on the next line compares the sales volume in *Volume* with *DiscountVolume.* If *Volume* is greater than *DiscountVolume,* the calculations following, down to the *Else* statement, are carried out. Otherwise, the calculation after the *Else* is carried out. If *Volume* is greater than *DiscountVolume,* the percent discount rate is found in the lookup table and assigned to the variable *DiscountPercent.* The invoice amount is then calculated by applying the full price to the units up to *DiscountVolume* plus the discounted price for units above *DiscountVolume.* Note the use of the underscore character, preceded by a blank space, to indicate the continuation of the code on the next line. The result is assigned to the name of the function, *InvoiceAmount,* so that the value will be returned to the worksheet cell. If *Volume* is not greater than *DiscountVolume,* the invoice amount is calculated by applying the price to the units sold and the result is assigned to the name of the function.

7. Return to the Excel spreadsheet, make cell C10 the active cell and type in the formula **=InvoiceAmount(A10, B10).**
8. Copy the formula to the remaining cells.
9. If all your variables and functions are correct, the invoice amounts should look like Figure M.13.

Congratulations, you have just completed your first VBA application!

Figure M.13
Example Figures



SUMMARY: STUDENT LEARNING OUTCOMES REVISITED

1. **Explain the value of using VBA with Excel.** Excel VBA is a programming application that allows you to use Visual Basic code to customize your Excel applications.
2. **Define a macro.** A macro is a series of steps that are completed automatically. For example, you can write a macro that inserts your name in a specific cell address and then prints the worksheet.
3. **Build a simple macro using a Sub procedure and a Function procedure.** Building a Sub procedure will perform only a procedure. Building a Function procedure will return a value.
4. **Describe an object.** Objects are items available to control your code, such as a workbook, a worksheet, a cell range, a chart, and a shape.
5. **Explain the difference between a comment, a variable, and a constant.** A comment is the simplest type of VBA statement that is used to clarify a piece of code you wrote. A variable is a named element that stores things, typically a value of something. A constant is a named element whose value doesn't change.
6. **List the various Visual Basic Application data types and operators.** There are many different data types that are used in Visual Basic Applications. The most common ones are: Boolean, Integer, Long, Single, Double, Date, String. The various operators used are +, - ,/,*, AND, OR, NOT.
7. **Describe and build a macro that uses the *If-Then-Else, For-Next, Do-Until, Do-While,* and *Select Case* structures.** Use the *If-Then* structure when you want to execute one or more statements conditionally. Use the *For-Next* loop if you know the iteration amount of the loop. Use a *Do-While* loop while a specified condition is met. Use a *Do-Until* loop until a condition is met. Use the *Select Case* structure for decisions involving three or more options.

KEY TERMS AND CONCEPTS

Constant, M.17
Function procedure, M.4
Looping, M.22
Macro, M.2
Macro language, M.2
Object, M.4
Sub procedure, M.3
Variable, M.5
Visual Basic Editor (VBE), M.5

ASSIGNMENTS AND EXERCISES

1. **AUTOMATING REPETITIVE TASKS** Once a week you have to develop a new worksheet for your department head that inserts enrollment data. More specifically, most of the tasks that you perform in creating the worksheet are very repetitive since the structure of the worksheet is always the same. You want to automate the steps that are repetitious. Using the Macro Recorder, create a macro that types six month names as three letter abbreviations, "Jan" to "Jun," across the top of a worksheet, starting in cell B1. Make each abbreviate bold, italics, and centered within each cell. Call the macro **MonthNames** and assign the macro the shortcut key **Ctrl+Shift+M.** Save the workbook as **MonthNames.xls.** Open a new worksheet, and press **Ctrl+Shift+M.**
2. **CALCULATE TAX VALUES** You are part of a programming team developing point-of-sale terminal software, but first you want to create a prototype of the logic this software needs to perform. For your prototype, you decide to create a macro function in Excel that will calculate the sales tax (4.9%) from the data file, **XLMM_SalesTax.xls.**
3. **DETERMINE SHIPPING CHARGES** Trans-Port Inc., a distribution company located in Denver, Colorado, needs some assistance in computing the shipping charges for freight. The shipping charge is calculated by the total weight of the shipment. Any shipment with a total weight of 500 or over is computed by taking the total weight and multiplying it by $1.00. Any shipment with a total weight of 100 pounds or more but less than 500 pounds is calculated by taking the total weight and multiplying it by .50. Anything shipped below 100 pounds is assessed a flat fee of $100. The owner of Trans-Port, Inc., Jake Plummer, has asked you to assist him in creating a macro function that will automatically assign shipping charges.
4. **ASSESS THE LETTER GRADE** You are a Teaching Assistant for the Information Technology department at your school. One of the professors, Dr. Hans Hultgren, has asked you to review a grading spreadsheet for him and write a macro function that will take the numerical score of each student and assign a letter grade. The grading scale is as follows:

Numerical Range	Letter Grade
90–100	A
80–89	B
70–79	C
65–69	D
<65	F

Dr. Hultgren has suggested that you write a macro that uses a *Select Case* statement. He has provided you with some mocked-up data, **XLMM_Grades.xls,** to use as a prototype, since giving you "real" grades is considered unethical.