

Chapter 1 - VBA

1.1 Introduction.....	1-2
1.2 Visual Basic for Applications with Excel.....	1-2
1.2.1 The VBA Integrated Development Environment (IDE).....	1-2
1.2.2 Programming Components within Excel.....	1-5
1.2.3 Getting Help with VBA	1-9
1.2.4 Constructing the Colorful Stats Program.....	1-10
1.3 Beginning Programs with VBA.....	1-13
1.3.1 Variables, Data Types, and Constants	1-13
1.3.2 Simple Input and Output with VBA	1-21
1.4 Procedures and Conditions	1-23
1.4.1 VBA Procedures	1-24
1.4.2 Manipulating Strings with VBA Functions	1-28
1.5 Procedures and Conditions	1-31
1.5.1 VBA Procedures	1-31
1.6 Loops and Arrays.....	1-46
1.6.1 Looping with VBA	1-46
1.6.2 ARRAYS	1-51
1.6.3 Programming Formulas into Worksheet Cells.....	1-58
1.6.4 R1C1-Style References.....	1-60
1.7 Basic Excel Objects	1-60
1.7.1 VBA and Object-Oriented Programming	1-61
1.7.2 VBA Collection Objects	1-61
1.7.3 EXCEL OBJECTS.....	1-63
1.7.4 The Worksheet Object	1-63
1.7.5 The Range Object	1-64
1.7.6 Working with Objects.....	1-66
1.8 Basic File I/O & Debugging	1-68
1.8.1 Debugging.....	1-68
1.8.2 File Input and Output (I/O).....	1-71
1.8.3 More on File Handling.....	1-79

CHAPTER 1 - (VBA)

Visual Basic for Applications

1.1 Introduction

Visual Basic for Applications (VBA for short) is a programming environment designed to work with Microsoft's Office applications (Word, Excel, Access, and PowerPoint). Components in each application (for example, worksheets or documents) are exposed as objects to the programmer to use and manipulate to a desired end. Almost anything you can do through the normal use of the Office application can also be automated through programming. VBA is a complete programming language, but you can't use it outside the application in which it is integrated. This does not mean VBA can be integrated only with Office programs. Any software vendor that decides to implement VBA can include it with their application.

VBA is relatively easy to learn, but to use it in a new application, you must first become familiar with the object model of the application. For example, the Document and Dictionary objects are specific to the Word object model, whereas the Workbook, Worksheet, and Range objects are specific to the Excel object model. As we proceed, you will see that the Excel object model is fairly extensive; however, if you are familiar with Excel, you will find that using these objects is generally straightforward.

1.2 Visual Basic for Applications with Excel

In this section we introduce you to the programming tools available in Excel. These tools include the VBA IDE (Integrated Development Environment), controls and functions available through the main Excel application, and VBA on-line help. After your introduction to the VBA programming environment, I take you through a very short and simple program that calculates some basic statistics from a sample data set. Specifically this section will cover:

- The VBA IDE and components within
- Programming tools within Excel
- Using VBA on-line help

1.2.1 The VBA Integrated Development Environment (IDE)

Before learning how to program in VBA, you have to learn how to use the software required for creating your projects. The VBA development software is included with each component of the Microsoft Office suite of programs, including Excel. Starting the VBA development software places you in the VBA programming environment IDE, which provides you with a number of tools for use in the development of your project.

1.2.1.1 Getting to the IDE from Excel

Before you begin creating projects with VBA you must know your way around the IDE. You can access the IDE from Excel in a couple of different ways. In Excel: select Tools, Macro, Visual Basic Editor (as shown in Figure 1.1); or use the keystroke Alt + F11.

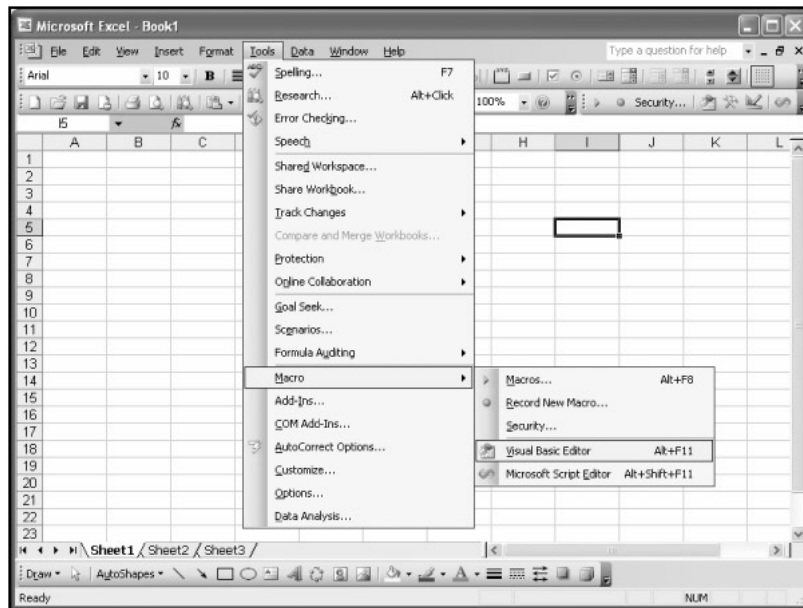


Figure 1.1 - Accessing the VBA IDE from the Tools menu in Excel.

Alternatively, select the Visual Basic toolbar from the View/Toolbars menu item in Excel. When the toolbar is displayed, select the Visual Basic Editor icon found in the middle of the toolbar (see Figure 1.2).

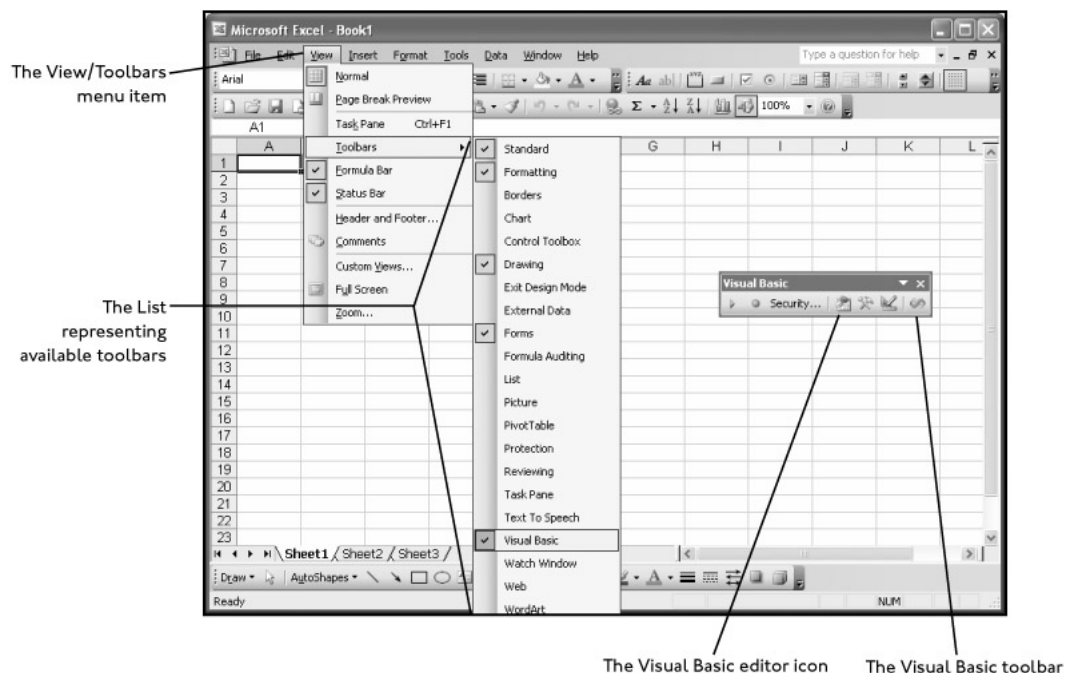


Figure 1.2 - Accessing the VBA IDE from the Visual Basic toolbar

1.2.1.2 Components of the IDE

After opening the VBA IDE you may find yourself looking at a window similar to what is shown in Figure 1.3. This figure shows the VBA IDE and some of the tools that can be used to create projects.

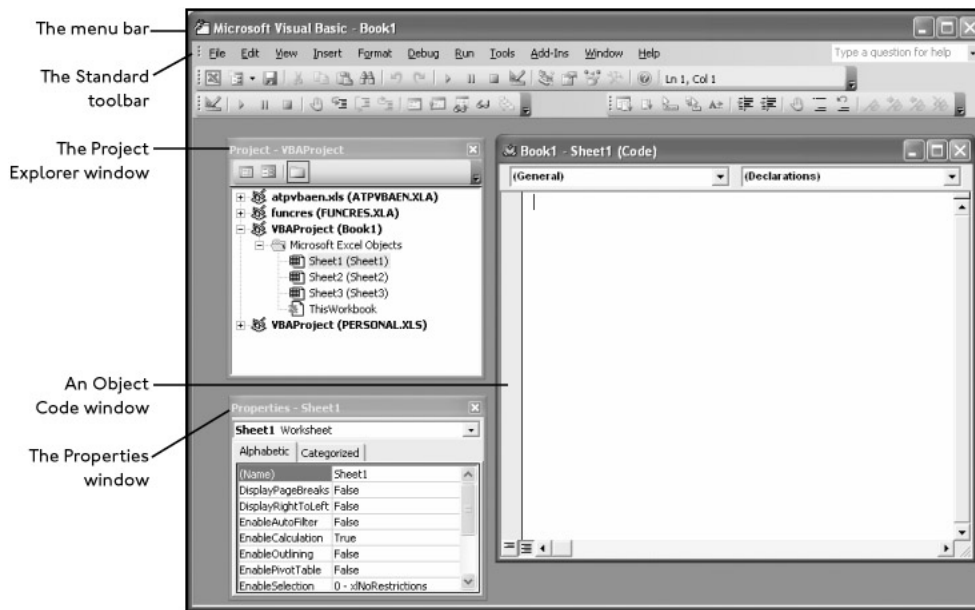
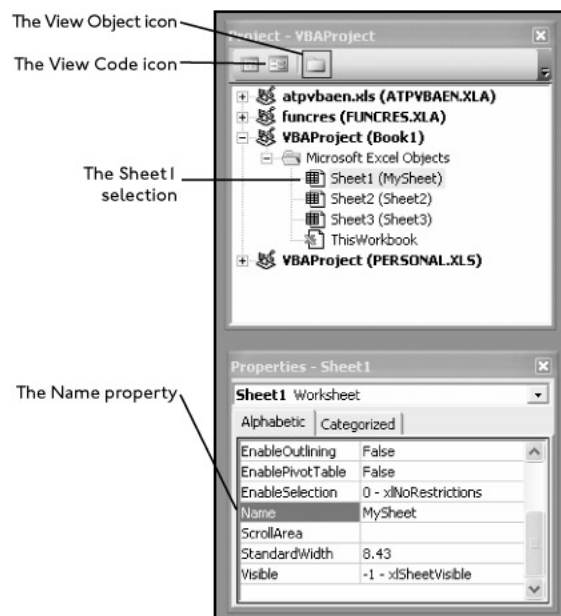


Figure 1.3 - Accessing the Project Explorer and Properties windows.

Once the Project Explorer window is displayed, find the project that represents the workbook you opened while in Excel (probably Book1 or Book2). If the components of the workbook you opened in Excel are not displayed, click the + sign next to the Microsoft Excel Objects folder directly underneath the project name. Next, find the object labeled Sheet1, select it with your mouse and then turn your attention to the Properties window. Scroll down the Properties window until you come to the Name property (the one without the parentheses around it). Delete the text entered to the right of the Name property and enter MySheet. Figure 1.5 illustrates how to find the Name property.



Toggle back to Excel by pressing Alt+F11, or select it from the taskbar in Windows. You will note that the name of Sheet1 has now been replaced with MySheet in your Excel workbook, as shown in Figure 1.4. See how easy it is to alter the properties of a worksheet in Excel using VBA? As VBA developers, however, we will seldom, if ever, alter the properties of a workbook or worksheet at design time.

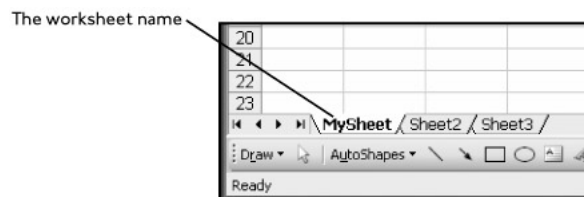


Figure 1.4 - An edited worksheet name in Excel.

Figure 1.5 - Accessing the Name property of a worksheet.

The bulk of the work affecting workbooks and worksheets will occur at run time; however, we will alter properties of ActiveX controls at design time.

HINT

Design time refers to project development and the manipulation of object properties using the VBA IDE prior to running any code. Conversely *run time* will refer to the manipulation of object properties using a program; thus, the properties of the object do not change until the code is executed.

Finally, I will show you one more component of the VBA IDE. If you look back at Figure 1.3 you will also see a standard code window. Windows such as these are used as containers for your program(s). This is where you type in the code for your program, so these windows are essentially text editors very similar to Notepad. You must be aware that there are pre-defined code windows for specific Excel objects, namely the workbook (for example, ThisWorkbook) and the worksheets (for example, Sheet1). The code window displayed in Figure 1.3 represents Sheet1 contained within the workbook Book1.

You will also be able to add components to your project and they will have their own code windows. I will explain how to use code windows more thoroughly as we proceed. For now, know that you can open a code window by double clicking on any object listed in the Project Explorer. You can also select the object in the Project Explorer and click on the View Code icon at the top left of the window (refer to Figure 1.5), select Code from the tools menu, or press F7 (refer to Figure 1.3). Note that you can also view the selected object in Excel by selecting the appropriate item from these same locations (refer to Figure 1.4 and Figure 1.5). There are, of course, more components to the VBA IDE, but I've shown you enough to get you started for now. As the need arises, I will introduce more tools from the IDE that will aid in the development of various projects.

1.2.2 Programming Components within Excel

Not everything of interest to the VBA programmer can be found in the VBA IDE. There are a few programming-related components that you can access from the Excel application. The components I am referring to are the Macro items found under the Tools menu, and three of the available toolbars—Visual Basic, Control Toolbox, and Forms—found in the View menu in Excel.

1.2.2.1 Macro Selection

Now that you've had an introduction to the VBA IDE, it's time to look at development tools accessed directly from Excel. To begin, take a closer look at the Macro selection from the Tools menu, shown in Figure 1.1. Notice two other items displayed in Figure 1.1 that I have not yet discussed: Macros and Record New Macro. Essentially the Record Macro tool will allow you to create a VBA program by simply selecting various tasks in Excel through the normal interface. The Macros menu item will simply display a dialog box with a list of some or all of the currently loaded VBA programs.

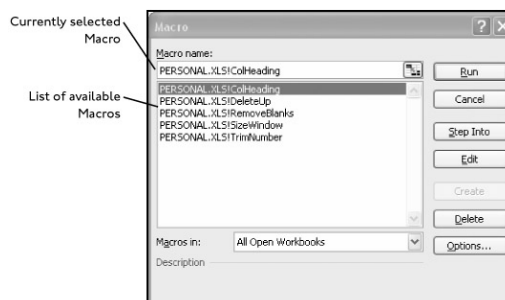


Figure 1.6 - The Macro dialog box displaying the available VBA programs

Macro menu item is one way to access and run desired VBA programs. Figure 1.6 shows the Macro dialog box.

HINT

Macros typically refer to programs that are recorded as the user executes a series of tasks from the normal application interface. They are useful when a user repeatedly performs the same tasks in Excel. Instead of having to repeat tasks, the user can simply record his/her actions once, then "play back" the macro when he/she needs to repeat the same series of tasks. However, it is possible to access programs that were not recorded through the Macro menu item, thus I will use the term macro to refer to both recorded programs and those programs written from scratch.

1.2.2.2 The Visual Basic Toolbar

The Visual Basic toolbar shown in Figure 1.2 provides another set of tools for the VBA developer. You have already seen how selecting the Visual Basic Editor icon from this toolbar gives you access to the VBA IDE. There are several other useful items on the Visual Basic toolbar, including Run Macro, Record Macro, and Design Mode. Also included on the Visual Basic toolbar is an icon for the Control Toolbox, denoted by the crossed hammer and wrench. The Control Toolbox can also be accessed via the Toolbars item on the View menu.

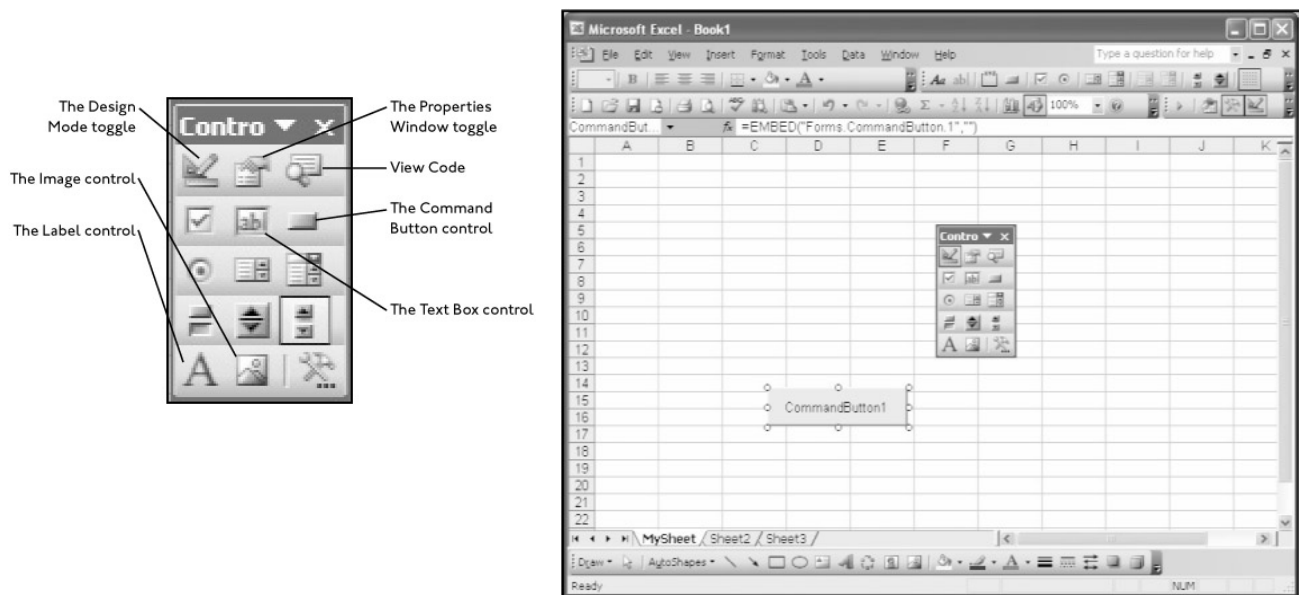


Figure 1.7 - The Control Tool Box

The Control Toolbox (refer to Figure 1.7) provides you with ActiveX controls which are graphical tools, such as a Check Box or Command Button, that may be associated with a macro. The Text Box, Command Button, Label, and Image Control are just some of the ActiveX controls available and are specifically labeled in Figure 1.7. You place controls on a worksheet by first clicking on the desired control and then drawing it onto the worksheet. Start by selecting the Command Button control and drawing it on a worksheet as shown in Figure 1.7.

After the Command Button is placed on the worksheet, you will notice that it is selected and the application is currently in Design Mode (check that the Design Mode icon in the upper left corner of the Control Toolbox appears "pressed in"). You can access the properties of the Command Button control while in Design Mode. With the Command Button control selected while in Design Mode, select the Properties icon from the Control Toolbox. A window much like the Properties window in

the VBA IDE will appear. The Properties window lists all the attributes or properties used to describe the Command Button control. Figure 1.8 shows the Properties window.

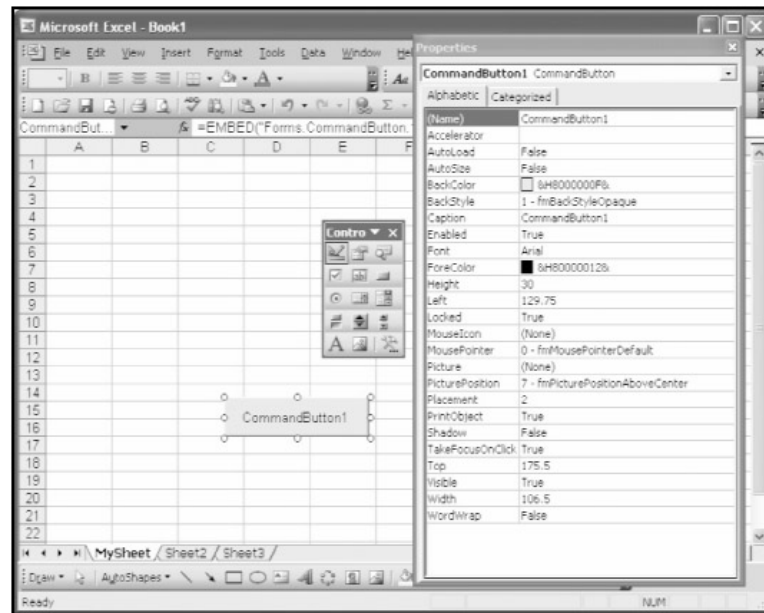


Figure 1.8 - The Properties window of the Command Button control.

In the Properties window of the Command Button control change the Caption property to Click Me and then notice how the new caption is displayed on the control. You should also change the Name property to something like cmdColorChange. The prefix cmd references the type of control (Command Button) and the rest of the name refers to the function of the program that is triggered when the button is pressed. You can also play with some of the other properties such as Font, ForeColor, BackColor, Width, and Height to change the appearance of the control. You can even display a picture within the Command Button control through the Picture property, and then select an image file from your computer.

TRICK

The Name property is an important property of any ActiveX control. The value of the Name property should be changed to something meaningful as soon as the control is added to the worksheet. Typically, an abbreviated word telling us the type of control (the cmd at the beginning of the name above denotes a Command Button) and its function in the program will work well. The Name property of an ActiveX control should be changed if you refer to it in your program. A meaningful name will help you remember it, as well as make the code more readable.

Once the appearance of your Command Button control is to your liking, select the View Code icon from the Control Toolbox, or double click on the Command Button control to access the code window. You will be taken immediately to the VBA IDE. Now it's time to make the Command Button control functional, and you can only do that by adding code to its code window. Figure 1.9 shows the code window for the Command Button control.

The title bar tells us the object to which this code window belongs. In this case, the code window belongs to the worksheet named Sheet1 in the workbook named Book1. This is because I placed the Command Button control on Sheet1 of Book1 in the Excel application. You may recall that I changed the name of the worksheet in Excel to MySheet, but the name of the worksheet as it will have to be referenced in code is still Sheet1. In the upper left corner of the code window is a

dropdown list box containing the names of all objects contained within the selected worksheet. The name of the Command Button control is displayed because the cursor in the editor is within an event procedure of this Command Button control.

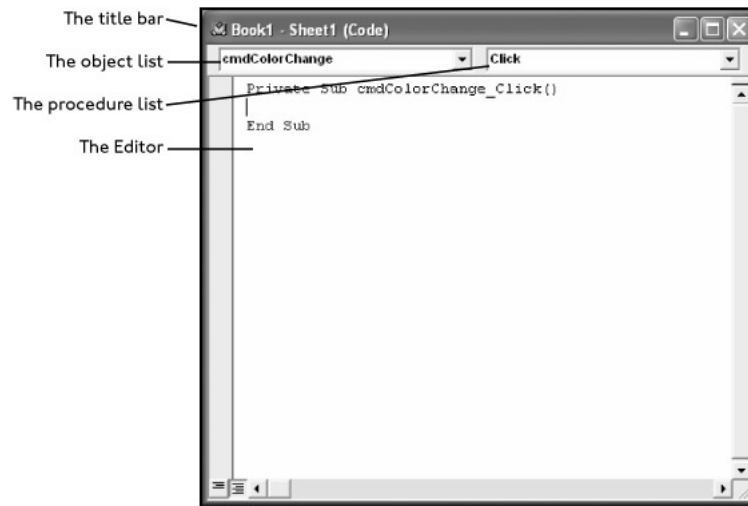


Figure 1.9 - The VBA IDE showing the code window for the worksheet named Sheet1.

HINT

Event procedures are self-contained blocks of code that require some type of stimulus in order to run. The stimulus often comes directly from the user (for example, a mouse click), but may also result from another piece of code.

Event procedures are predefined for ActiveX controls and other Excel objects, such as workbooks and worksheets. All event procedures for the selected object are listed in the upper right corner of the code window in a dropdown list box. The Click() event is a very common event procedure that is built into most ActiveX controls. Any code placed within the predefined procedure will trigger when the user clicks once on the object—in this case, the Command Button control named cmdColorChange. The procedure is defined as listed in Figure 1.9 with the following two lines of code:

```
Private Sub cmdColorChange_Click()  
End Sub
```

The name of the procedure will always be the name of the object with an underscore followed by the name of the event. You cannot change the name of a predefined event procedure without changing the Name property of the object. If you do change the name of the event procedure, the code within the procedure will not run when you want it to. The keyword Sub is required and is used as the defining opening of any procedure—event-type or programmer-defined. Private is an optional keyword; The second line End Sub is always used to close a procedure. Now type the following lines of code within the Click() event procedure of the Command Button control named cmdColorChange.

```
Range("A1").Select  
Cells.Interior.ColorIndex = Int(Rnd * 56) + 1
```

These two lines will select cell A1 on the worksheet and set the fill color of all cells in the worksheet to one of fifty-six possible colors. This is the equivalent of a user first selecting all the cells in a worksheet and then changing the fill color from the formatting toolbar in the Excel application. The color of the cells is chosen randomly and will change with each click of the

Command Button control because the above code will run once with each click event. So the entire procedure now looks like the following.

```
Private Sub cmdColorChange_Click()  
    Range("A1").Select  
    Cells.Interior.ColorIndex = Int(Rnd * 56) + 1  
End Sub
```

Return to the Excel application and exit Design Mode by toggling the icon on the Control Toolbox (refer to Figure 1.7). Now test the program by clicking on the Command Button control. The color of all cells in the worksheet will change color with each click. Figure 1.10 shows an example of my worksheet after one click on the Command Button control.

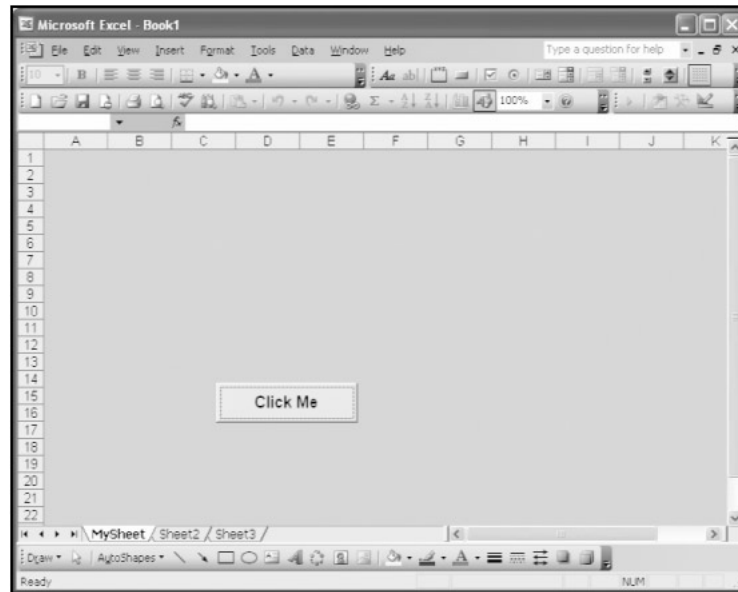


Figure 1.10 - The Color Changer program.

You can save the workbook as you would an Excel workbook. The Command Button control and event procedure code will be saved with the workbook.

1.2.3 Getting Help with VBA

I can't emphasize enough how important it is that you become comfortable with the on-line help in the VBA IDE (not to mention in the Excel application). The on-line help provides fast access to solutions for any programming problems you have with your project. Books make good resources and are much better at teaching you how to program, but they can't cover everything. Often, all you need to see is a simple example of how to use a particular function or other keyword; the on-line help does contain documentation on every keyword, programming construct, and object you might use in your project. The bottom line is this: there is always something helpful on-line, it's just a matter of finding the right document.

To access the VBA help, you must have the IDE open and active; otherwise, everything is the same, from the Help menu to the help window and even the office assistant (if you choose to use it). Select Help, Microsoft Visual Basic Help to activate the Visual Basic Help dialog box. With the Visual Basic Help dialog you can browse a table of contents or enter keywords to search for on-line documentation. After you select a topic, documentation related to that topic appears in another Visual Basic Help window.

1.2.4 Constructing the Colorful Stats Program

When starting a project, programmers often compile a list of specific requirements, then refer to this list while designing the algorithm(s) that will be followed when writing the program. The advantage you have when compiling a requirement list is that the source document can be used to build a protocol for testing the program.

Requirements of the Colorful Stats Program

The purpose for the Colorful Stats program (as it relates to this book) is to give you a demonstration of ActiveX controls, event procedures, and using VBA to interact with an Excel worksheet. The practical purpose of the *Colorful Stats* program is to allow a user to immediately calculate basic statistics for a selected set of data. I've defined a few specific requirements for the *Colorful Stats* program and they are listed as follows:

1. The program shall calculate the following statistics for a selected data set—the number of data elements selected by the user, the minimum value, the maximum value, the sum total, the average value, and the standard deviation.
2. The program shall use Excel worksheet formulas to calculate the statistical parameters listed in Requirement 1.
3. The program shall write the formulas for the statistical parameters to the worksheet cells D2 through D7. Corresponding labels shall be written to cells C2 through C7.
4. The program shall change the interior color of cells C2 through D7 to green.
5. The program shall change the border color of cells C2 through D7 to red.
6. The program shall format the font of cells C2 through D7 to Arial, 16 pt, bold, and blue.
7. The program shall be initiated from a mouse click of a Command Button control placed on the worksheet.

Designing the Colorful Stats Program

When designing a program, I consider the user interface, program inputs and outputs, the location of the code (for example, event procedures of ActiveX controls), and the use and configuration of other programming components that I have not yet discussed. I start by making the very simple user interface for the *Colorful Stats* program.

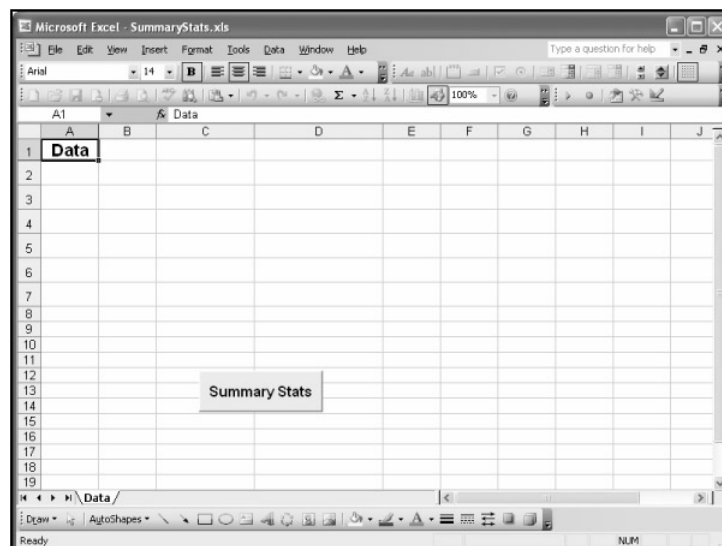


Figure 1.11- The user interface for the *Colorful Stats* program.

The interface will use a single Command Button control placed on a worksheet to activate the program. I'm assuming that the data will be entered in column A of the worksheet (although this is not required) so I will place the Command Button control in columns C and D, close enough to the top of the worksheet so it is likely to be seen by the user when opened, but below row 7 to avoid masking the statistical values (refer to Figure 1.11). Note that I have altered the Name, Caption, and Font properties of the Command Button control.

All program inputs and outputs are from, and to, the current active worksheet. The data used in the calculation of the statistical values must come from the cells that are selected by the user. I will write the program to output cell formulas to the desired worksheet cells so that Excel calculates the statistical values. I must also output labels to the cells adjacent to the statistical values for clarity. I will also format all output as described in the requirements. Finally, the program is to be initiated from a user's click of the Command Button control, so I will enter all programming statements in the Click() event procedure of the Command Button control.

TRICK

Ideally, the *Colorful Stats* program would be activated from an interface independent of the worksheet that contains the data (i.e., using an ActiveX control on the worksheet containing the data is not the best solution). The program should also write the statistics to a new worksheet rather than risk overwriting data in the active worksheet. However, this requires a little more programming than I should show you right now. At this point in the book, the only tool I've shown you for running a loaded macro that may be independent of the selected worksheet is the Macro dialog box (refer to Figure 1.6). As you proceed through this book you will learn other methods for initiating macros and how to create new worksheets

Coding the Colorful Stats Program

All of the code is to be placed in the Click() event procedure of the Command Button control. The code window can be accessed via the VBA IDE by double clicking on the Command Button control while in Design Mode. You can also select the appropriate object (cmdCalculate) from the object dropdown list in the code window for the worksheet on which the ActiveX control was placed (refer to Figure 1.12).

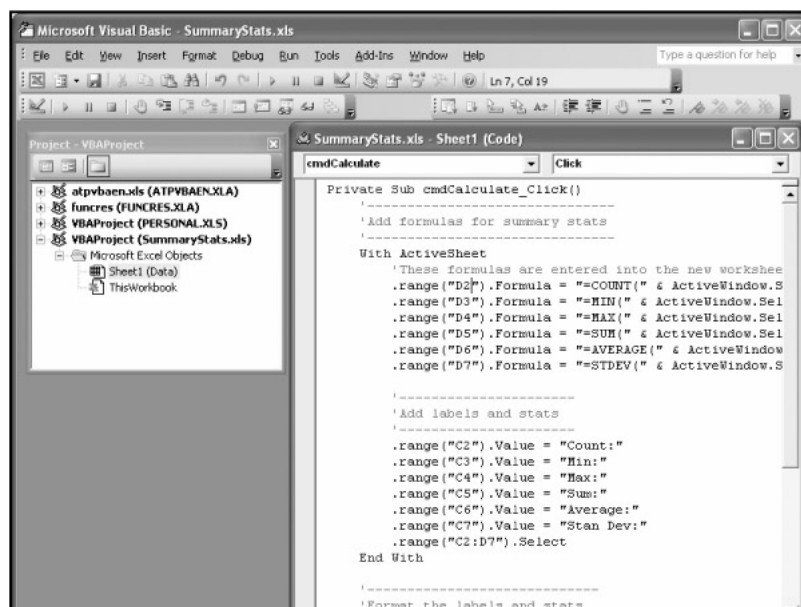


Figure 1.12 - VBA IDE showing the code window for the worksheet

As you can see, the following code was placed in the Click() event procedure of the cmdCalculate Command Button control. Now let's take a closer look at each line of code. The very first and last lines define the type of procedure as a Click() event, as described earlier in this Section. Immediately following the opening line of code is a comment.

HINT

Comments (or remarks) are notes left in the code by the programmer to help describe the function of the program. Comments make it easier to find problems with the code, or add different features to the code at a later time. Enter comments (also known as remarks) into the code by beginning the line with an apostrophe (or Rem). You must enter another apostrophe for each new line; the VBA text editor will color each comment line green (default color; change by selecting Tools, Options, Editor Format, and Comment Text from the list of Code colors). Comments are not part of the program, and are ignored when the program runs; thus, comments do not decrease the execution speed of a program.

```
Private Sub cmdCalculate_Click()
    '-----
    'Add formulas for summary stats
    '-----
    With ActiveSheet
        'These formulas are entered into the new worksheet.
        .range("D2").Formula = "=COUNT("& ActiveWindow.Selection.Address & ")"
        .range("D3").Formula = "=MIN("& ActiveWindow.Selection.Address & ")"
        .range("D4").Formula = "=MAX("& ActiveWindow.Selection.Address & ")"
        .range("D5").Formula = "=SUM("& ActiveWindow.Selection.Address & ")"
        .range("D6").Formula = "=AVERAGE("& ActiveWindow.Selection.Address & ")"
        .range("D7").Formula = "=STDEV("& ActiveWindow.Selection.Address & ")"
        '-----
        'Add labels and stats
        '-----
        .range("C2").Value = "Count:"
        .range("C3").Value = "Min:"
        .range("C4").Value = "Max:"
        .range("C5").Value = "Sum:"
        .range("C6").Value = "Average:"
        .range("C7").Value = "Stan Dev:"
        .range("C2:D7").Select
    End With

    '-----
    'Format the labels and stats.
    '-----
    With Selection
        .Font.Size = 16
        .Font.Bold = True
        .Font.Color = vbBlue
        .Font.Name = "Arial"
        .Columns.AutoFit
        .Interior.Color = vbGreen
        .Borders.Weight = xlThick
        .Borders.Color = vbRed
    End With
    range("A1").Select
End Sub
```

I will discuss code structures, Excel objects, and object syntax in subsequent Sections. If you are even somewhat familiar with Excel, however, you probably have a pretty good idea as to what's happening in the above code. First, the cell formulas are written to the indicated cells (D2 through D7) using the range selected by the user as the parameter for each worksheet function. Next, the statistical labels are written to the corresponding cells in the adjacent columns (C2 through C7). The last part of the program formats the font, border, and color of cells C2 through D7 before selecting cell A1. Another example of the worksheet after some arbitrary data has been entered in column A and the program run is shown in Figure 1.13. That's all there is to it! This code will run once each time the Command Button control is clicked (don't forget to exit Design Mode and select some data first).

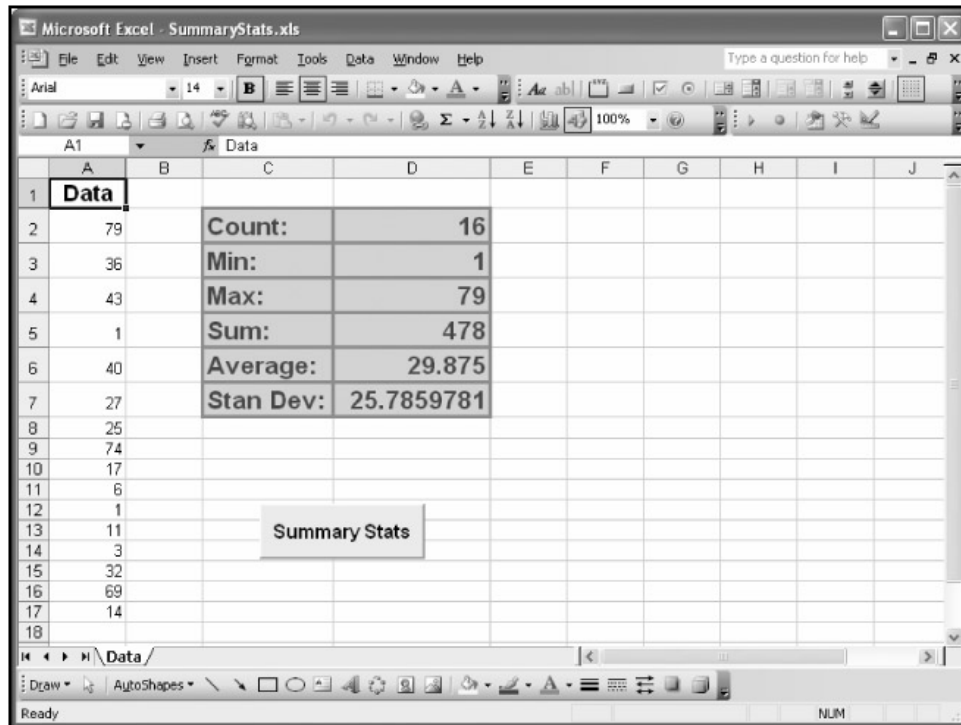


Figure 1.13 - The *Colorful Stats* program after running.

1.3 Beginning Programs with VBA

Now that you know your way around the VBA IDE for Excel, it's time to introduce some basic programming concepts common to all languages. The next three sections are devoted to these basic programming structures that, although they may not be that exciting, are essential for developing VBA projects.

Specifically, in this section we look at:

- Variables and data types
- Constants
- Simple input and output
- String functions

1.3.1 Variables, Data Types, and Constants

Since we focus on spreadsheet applications, it's only natural that I introduce variables by asking you to think about the following: what types of values can be entered into a spreadsheet cell and how

you might use them? You know that you can enter numbers and text in any spreadsheet cell in Excel. Also, you may or may not know that the format of a spreadsheet cell can be changed to one of several possibilities. For example, a number can be formatted such that the value is displayed with or without digits to the right of the decimal point. Numbers can also be formatted as currency or as a percentage (along with a few other options). Text can be displayed as entered or be automatically converted to a date or time. The content or value of a spreadsheet cell can be changed or deleted at any time.

HINT

From this point forward, the contents of a spreadsheet cell (text or numbers) in Excel will be referred to as its value.

In essence, spreadsheet cells are temporary storage containers for numbers and text that can be displayed and used in a number of different formats. This also describes a variable in any programming language. You can use variables in programs for temporary storage of data. For example, any data input by a user (possibly from a Text Box Control), can be stored in a variable and used later in the program. In the *Colorful Stats* project from section 1.2.4, the following line of code acts a lot like a variable.

```
.range("C6").Value = "Average:"
```

Here the text "Average" is copied to spreadsheet cell C6. I could have just as easily copied the text into a program variable first and then copied the contents of the variable to the cell C6. I didn't use an additional program variable because I wanted to save a couple of steps and because, as discussed earlier, spreadsheet cells already act a lot like variables. To accomplish this same task using a program variable, use the following:

```
Dim myString as String  
myString = "Average:"  
.range("C6").Value = myString
```

The variable myString is first declared assigned the string literal "Average:". The value of spreadsheet cell C6 is then assigned the value stored in the variable myString.

1.3.1.1 Declaring Variables

To declare a variable is to tell the computer to reserve space in memory for later use. To declare a variable use a Dim (short for Dimension) statement.

```
Dim myVar As Integer
```

The name of the variable is myVar. The name must begin with an alphabetic character and cannot exceed 255 characters or contain any spaces. You should avoid the use of punctuation marks or other unusual characters in the variable name, as many of them are not allowed; however, the underscore character *is* allowed and works well for separating multiple words contained within a single variable name (for example, First_Name). Avoid using reserved VBA keywords and don't repeat variable names within the same scope (discussed later in this chapter). As a convention, the variable name should be descriptive of the value it will hold. For example, if you use a variable to hold someone's first name, then a good name for that variable might be firstName or FirstName. My preference is to begin a variable name with a lowercase letter and then capitalize the first letter of any subsequent words appearing in the name. I try to keep the length to a minimum (fewer than 12 characters) only because I don't like typing long names. Of course, you can adopt your own conventions as long as they don't contradict rules established by VBA.

TRICK

Use Option Explicit in the general declarations section of a module window to force explicit variable declarations (see Figure 1.14 and Figure 1.15). Otherwise variables can be dimensioned implicitly (without a Dim statement) as they are required in code. In other words, you can begin using a new variable without ever declaring it with a Dim statement if you don't use the Option Explicit statement. This is not good programming practice as it makes your code harder to interpret, and subsequently more difficult to debug. You can automatically have Option Explicit typed into each module window by checking the Require Variable Declaration option in the Tools/Options menu item of the VBA IDE.

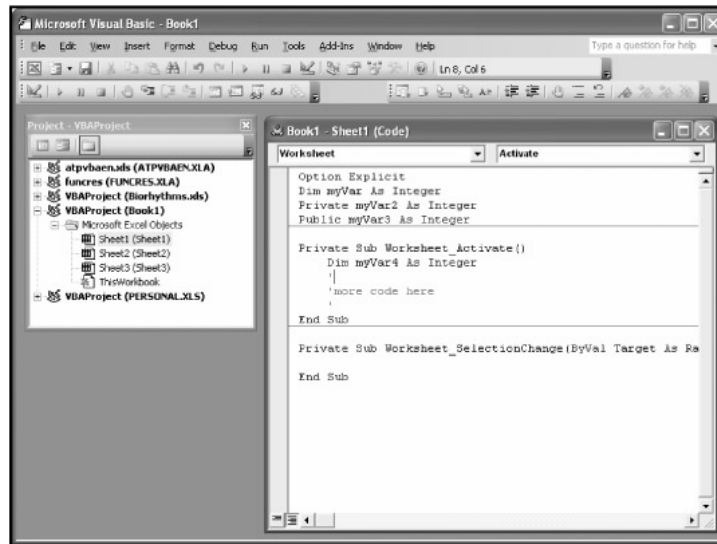


Figure 1.14 - The object module for an Excel worksheet.

Following the variable name, the data type is specified for the variable. In the example above, the variable is declared as an integer data type. This tells VBA what kind of data can be stored in this variable and how much memory must be reserved for the variable. I will discuss data types in detail later in this chapter.

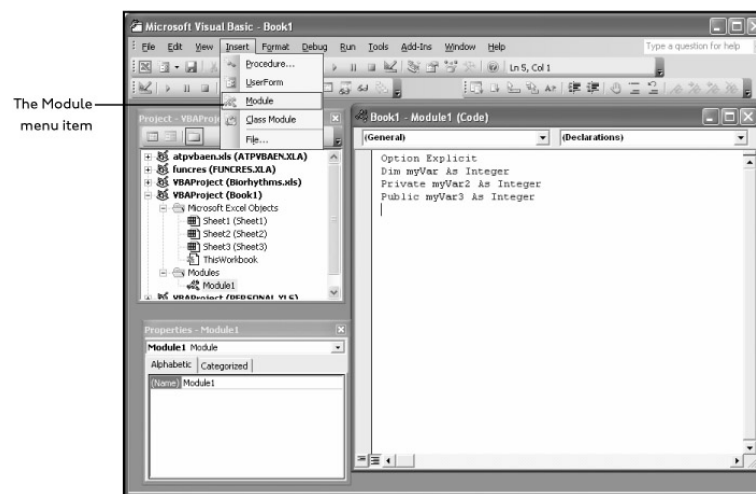


Figure 1.15 - Inserting a standard module.

1.3.1.2 Object and Standard Modules

Modules refer to a related set of declarations and procedures. Each module will have a separate window in the VBA IDE and, depending on the origination of the module, it will have different behavior with regard to variable declarations. I will refer to the module window shown in Figure 1.14 as an *object* module because it is associated with an object (the Worksheet object in this example).

This module will automatically contain all event procedures associated with the worksheet Sheet1, and any ActiveX controls added to this worksheet. Object modules may also contain programmer-defined procedures. Each worksheet will have a separate code window as will the workbook. A standard module must be added to the project via the Insert menu of the VBA IDE, as shown in Figure 1.15. Standard modules are contained within a separate folder in the Project Explorer and may be renamed in the Properties window (see Figure 1.15). Standard modules contain variable declarations and programmer-defined procedures.

1.3.1.3 Variable Scope

Scope, in the context of variables, refers to the time when a variable is visible or available to the program. When a variable is in its scope, it can be accessed and/or manipulated. When a variable is out of scope, it is unavailable—essentially invisible to the program.

A variable declared within the code block of a procedure (such as the Click() event procedure of the Command Button control), is a *procedural level* variable. Procedural level variables are only available while program execution occurs within the procedure that the variable was declared. In Figure 1.14, the variable myVar4 is only visible to the program while the code in the Activate() event procedure of the worksheet executes. When program execution is triggered by the Activate() event, the variable myVar4 is dimensioned in memory. Program execution proceeds through the event procedure until reaching the End Sub line of code, after which the variable is released from memory and is no longer available. Each time the procedure executes, the variable is created and destroyed. Thus, myVar4 will not retain its value between calls to the procedure. If necessary, the Static keyword can be used to tell VBA to remember the value of the variable between calls to a procedure. Consider the following example:

```
Private Sub Worksheet_Activate()  
    Static myVar4 As Integer  
    myVar4 = myVar4 + 1  
End Sub
```

In this procedure the variable myVar4 will increment its value by one with each call to the procedure. If you replace the Static keyword with Dim, myVar4 will never exceed a value of 1.

TRICK

Integer variables are initialized to a value of 0 at declaration.

Declaring a variable outside of a procedure with a Dim statement makes it a *module level* variable. The scope of a module level variable depends on the keyword used in the declaration. For example in Figure 1.14 the variables myVar, myVar2, and myVar3 are declared outside all procedures.

HINT

The area outside of any defined procedure is known as the *general declarations* section of a module (object or standard). This area can only be used for declarations.

These three variables are declared with the Dim, Private, and Public keywords. The Private and Public keywords are only allowed for variable declaration in the general declarations section of a module. Each of the three variables, myVar, myVar2, and myVar3 are visible to any procedure within this module. In addition, the variable myVar3 is visible to any procedure in any module of this project. Variables declared in the general declarations section of a module (object or standard) with the Public keyword are commonly referred to as *global*.

TRICK

When declaring a variable with the Public keyword in the general declarations section of an object module, it must be referenced in other modules of the project by first identifying the name of the object module. For example, to reference and assign a value to the variable myVar3 in Figure 1.14 in any other module in that project, you must use code similar to the following:

```
Sheet1.myVar3 = 5
```

You do not have to reference the name of the module for variables declared with the Public keyword in the general declarations section of a standard module.

To summarize: the keywords Dim and Private have the same function in variable declarations when used in the general declarations section of any module; the Public keyword can be used to declare global variables in a standard or object module.

1.3.1.4 Data Types

Data types define the kind of value that may be stored within the memory allocated for a variable. As with spreadsheet cells, there are numerous data types; the most common are defined in Table 1.1.

Table 1.1 - Common VBA Data Types

DATA TYPE	STORAGE SIZE	RANGE
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,647
Single (floating-point)	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double (floating-point)	8 bytes	-1.79E308 to - 4.94E-324 for negative values; 4.94E-324 to 1.79e308 for positive values
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any Object reference
String (variable-length)	10 bytes + string length	0 to approximately 2 billion
String (fixed-length)	Length of string	1 to approximately 65,400
Variant (with numbers)	16 bytes	Any numeric value up to the range of a Double
Variant (with characters)	22 bytes + string length	Same range as for variable-length String
User-defined (using Type)	Number required by elements	The range of each element is the same as the range of its data type.

1.3.1.5 Numerical Data Types

The numerical data types listed in Table 1.1 are integer, long, single, and double. A variable declared as an integer or long data type can hold whole numbers or non-fractional values within the specified ranges. If you need a variable to hold fractional or "floating point" values, then use a single or double data type. Pay attention to the value of the number that might have to be stored within the variable. If the value gets too large for the data type, your program will crash. For example, the following code will generate an overflow error because the value 50000 is outside the allowed range for an integer data type:

```
Dim myNum As Integer
myNum=50000
```

You must also be careful about mixing numerical data types because you may not get the desired result. The following code will execute without errors, but the variable answer will hold the value 32 after execution of this block, not 31.8 as you might want.

```
Dim answer As Integer
Dim num1 As Single
Dim num2 As Integer
num1 = 5.3
num2 = 6
answer = num1 * num2
```

Changing the variable answer to a single data type will correct the problem. Using the code as shown above is a good way to ensure an integer is stored within a variable that receives its value from a computation involving floating point numbers. Notice that the value stored in answer is rounded to the nearest whole integer.

By using variables with numerical data types, you can carry out mathematical operations as you normally would using just the numbers the variables contained. You can add, subtract, multiply, and divide variables; you can square and cube numerical variables or raise them to any desired power. See Table 1.2 for a list of the operators used for common mathematical operations in VBA.

Table 1.2 - Mathematical Operators Used In VBA

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponential	^

Basically, any mathematical operation that can be performed on a number can be performed on a numerical variable. The following are a few examples:

```
Dim num1 As Integer
Dim num2 As Integer
Dim answer As Integer
num1 = 10
num2 = 5
answer = num1 + num2      ' answer Holds 15
answer = num1 - num2      ' answer Holds 5
answer = num1 * num2      ' answer Holds 50
answer = num1 / num2      ' answer Holds 2
answer = num1 ^ 2          ' answer Holds 100
answer = 2 ^ num2          ' answer Holds 32
```

After declaring the variables num1, num2, and answer, a few mathematical operations are carried out over several lines of code. The result of each line is given as a comment within the same line of code. In the code above, the equal sign (=) does not designate equality; instead it works as an assignment operator. For example, the variable answer gets the result of adding the two variables num1 and num2.

TRICK

Although it is not required, it is a good idea to place all variable declarations for a procedure at the start of your code. With variable declarations at the beginning of your code, you will be able to find them quickly when you need to debug.

1.3.1.6 String Data Types

Variables with string data types are used to hold characters as text. The characters can be numbers, letters, or special symbols (for example, punctuation marks). Basically, just about anything you can type on your keyboard can be held within a string variable. To declare a variable with the string data type, use the String keyword. To initialize a string variable, place the string value within double quotation marks.

```
Dim myText As String
myText = "VBA is fun"
```

There are two types of string variables, variable length and fixed length. The example above is that of a variable length string because myText can hold just about any length of text (see Table 1.1). Following is an example of a declaration for a fixed length string:

```
Dim myString As String * 8
myString = "ABCDEFGHijkl"
```

In the example above, the string variable myString can hold a maximum of eight characters. You can try to initialize the variable with more characters (as was done above), but only the first eight characters in this example will be stored in the variable. The value of myString is then "ABCDEFGH". Fixed length strings are more commonly used as a part of a user-defined data type discussed in a later chapter. In most cases, you will not know the length of the string to be stored in a variable so you should use the variable length type.

1.3.1.7 Variant Data Types

Variant data types are analogous to the General category in the number format of a spreadsheet cell in the Excel application. Variables are declared as variants by using the keyword Variant, or by not specifying a data type.

```
Dim myVar
Dim myVar2 As Variant
```

Variant type variables can hold any type of data except a fixed length string. Variant data types relax the restrictions on the value a particular variable can hold and thus give the programmer more flexibility; however, variant data types can also be dangerous if overused—they can slow down program execution—and programs with a large number of variant data types can be very difficult to debug. So while I don't recommend using them, I do recognize that many programmers do use variants, and the on-line help is filled with examples using variants, so I will offer a brief example here:

```
Dim myVar As Integer
myVar = 10
```

```
myVar = "Testing"
```

The example above will generate a type mismatch error because an attempt is made to enter the string "Testing" into an integer variable; however, if you change the variable myVar to a variant, the code will execute and myVar will hold the string value "Testing" when all is complete. The following code will run without error.

```
Dim myVar  
myVar = 10  
myVar = "Testing"
```

Using variants allows you to use the same variable to hold multiple data types (one at a time). The variable myVar holds the integer value 10 (albeit briefly) before being assigned the string value "Testing". You are probably starting to see the danger of using variant data types. Imagine a large program with numerous procedures and variables. Within this program are two variables of type variant that initially hold numerical values and will need to be used within the same mathematical operation before the program is finished executing. If one variable is mistakenly reinitialized with a string before the mathematical operation, an error will result and may crash the program (or at least taint the result). Debugging this program may present problems that depend on how hard it is to find the string initialization of the variant variable, and additional problems associated with the string variant. So even though it may be tempting to use variants as a way to prevent errors that crash your program (as in the example above), in actuality the use of variants make your code "loose," and may result in logic errors that are difficult to find.

HINT

Logic errors are the result of a mistake in a programming algorithm. They may or may not cause your program to crash, depending on the specific nature of the error. Trying to multiply variables of a string and integer data type would crash program execution, making the error relatively easy to find. Adding when you should have multiplied is a type of logic error that will not crash a program, but will certainly taint the result. Logic errors can be very serious because you may never find them or even know they exist.

1.3.1.8 Other Data Types

There are just a couple more data types that need to be mentioned. You will see them in action in subsequent chapters. The Boolean data type holds the value true or false. You can also represent true as a 1 and false as a 0. Boolean variables will be very useful when dealing with programming structures that use conditions. Declare and initialize a Boolean variable as follows:

```
Dim rollDice As Boolean  
rollDice = False
```

You can also specify variables of type date. Variables of type date are actually stored as floating point numbers with the integer portion representing a date between 1 January, 100 and 31 December 9999, and the decimal portion representing a time between 0:00:00 to 23:59:59. The date data type is mostly a convenience when you need to work with dates or times. There are a handful of VBA functions that use variables of type date that add to this convenience. You will see a couple of examples of date functions in the chapter project.

Constants

Constants allow you to assign a meaningful name to a number or string that will make your code easier to read. This is analogous to using named ranges in your spreadsheet formulas. There are numerous mathematical constants for which it makes sense to use constant data types. A constant

string might be used when you need frequent use of a particular spreadsheet label. Constants are declared using the Const keyword as shown below.

```
Const PI = 3.14159
Dim circumference As Single
Dim diameter As Single
diameter = 10.32
circumference = PI* diameter
```

The declaration and initialization of a constant occur in the same line of code. The value of a constant can never change, so it is a good idea to use constants when you need the same value throughout the life of your program. Constant names are uppercase as a convention only; it is not required by VBA.

1.3.2 Simple Input and Output with VBA

You have already seen how to get input from the user through the use of the Value property of a spreadsheet cell. Conversely, you can generate output for the user through the spreadsheet. Yet there may be times when you want something more dynamic and dramatic than a spreadsheet cell. The easiest method for gathering input from the user and sending output back is the InputBox() and MsgBox() functions.

HINT

Just as Excel comes with a large number of functions for the user to use in spreadsheet formulas (for example, the SUM() function), VBA contains numerous functions for the programmer. VBA programming functions, just like Excel functions, typically require one or more values (called parameters or arguments) to be passed to them, and then return one or more values (most commonly one) back to the program.

1.3.2.1 Collecting User Input with InputBox()

When you need to prompt the user for input and want to force a response before program execution continues, then the InputBox() function is the tool to use. The InputBox() function sends to the screen a dialog box that must be addressed by the user before program execution proceeds. Figure 1.16 shows the dialog box.

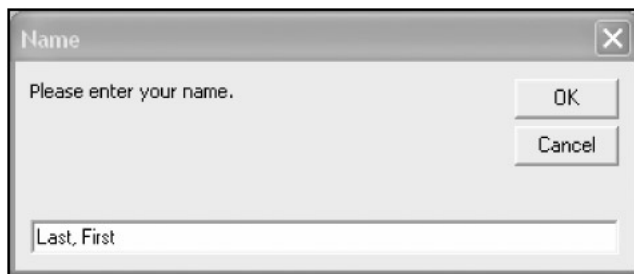


Figure 1.16- The InputBox() dialog box.

The InputBox() function returns the data entered by the user as a string if the OK button is clicked or the Enter key is pressed on the keyboard. If the user clicks the Cancel button, then a zero-length string is returned (""). Here is the syntax required for creating an InputBox() (parameters in brackets are optional).

```
InputBox(prompt [,title] [,default] [,xpos] [,ypos] [,helpfile, context])
```

The prompt is the only required parameter that must be passed to the function. Typically, the prompt, title, and sometimes the default are used. You must assign the return value of the function to a variable of type string.

```
Dim name As String
name = InputBox("Please enter your name.", "Name", "Last, First")
```

The prompt and title must be strings, which is why they are enclosed in double quotation marks. Alternatively, you can use string variables for these parameters. The title parameter is displayed in the title bar of the dialog box. The default parameter is displayed in the text box of the dialog box. Including a little help in the prompt or default parameter will increase the chances of getting the correct input. In the example above, I included a default parameter that serves to tell the user what format I want the name entered.

1.3.2.2 Output with MsgBox()

The MsgBox() function outputs a message to the user in the form of a message box like the one shown in Figure 1.17



Figure 1.17 - The message box.

Everything entered by the user is stored in the string variable userName.

```

Private Sub cmdBegin_Click()

    Dim userName As String
    Dim firstName As String
    Dim lastName As String
    Dim strLength As Integer
    Dim spaceLoc As Integer

    '-----
    'Collect user name, find the space between
    'first and last names, and separate the names.
    '-----
    userName = InputBox("Enter your first and last name.", "Name")
    spaceLoc = InStr(1, userName, " ")
    firstName = Left(userName, spaceLoc - 1)

    '-----
    'Output to the worksheet
    '-----
    Range("C3").Value = firstName
    strLength = Len(firstName)
    Range("C4").Value = strLength      'length of first name
    strLength = Len(userName)
    lastName = Mid(userName, spaceLoc + 1, strLength - spaceLoc)
    Range("C5").Value = lastName
    strLength = Len(lastName)
    Range("C6").Value = strLength
    Range("C7").Value = UCase(userName)
    Range("C8").Value = LCase(userName)
    Range("C9").Value = StrConv(userName, vbProperCase)
    Range("C10").Value = StrReverse(userName)
    Range("C11").Value = lastName & ", "& firstName
End Sub

```

To help picture what will happen in the program, let's assume the variable `userName` contains the string "Fred Flintstone". This string is 15 characters long; Table 1.4 shows the locations of each character.

Table 1.4- Character locations in a string

Character	F	r	e	d		F	l	i	n	t	s	t	o	n	e
Location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The program determines the location of the space by using the `InStr()` function. The `InStr()` function is passed three parameters, the number 1, the string variable `userName`, and a single character string containing a space. The parameter 1 represents the location to start searching within the string passed in the next parameter, in this case, `userName`. The last string is a space and this represents the character the `InStr()` function is searching for within the value of `userName`. The `InStr()` function then returns an integer value representing the location of the space within the `userName` string. This integer value is the location of the space between the first and last name of the user—in this example, location 5 (see Table 1.4)—and is stored in the integer variable `spaceLoc`. The `Left()` function is then passed two parameters, the `userName` string, and the length of the portion of the `userName` string to return. The variable `spaceLoc` is holding the location of the space (5 in our example), so using `spaceLoc - 1` for the length parameter in the `Left()` function returns just the first name ("Fred"). The `Len()` function is used to return the length of the `firstName` string as an integer and this value is stored in the variable `strLength`. The values of the `firstName` string and `strLength` variables are then copied to the worksheet.

The `Mid()` function is used to return the last name of the user to the string variable `lastName`. The `Mid()` function takes three parameters: the original string `userName` ("Fred Flintstone"), the starting location of the new string (`spaceLoc - 1`), and the length of the string to return (`strLength - spaceLoc`). The variable `strLength` was reinitialized to the length of `userName` prior to using the `Mid()` function. Again, the variables holding the last name and the number of characters in the last name are copied to the worksheet.

The `UCase()` and `LCase()` functions convert the `userName` string to all uppercase and all lowercase letters, respectively; and the `StrConv()` function converts the `userName` string to proper case. Finally, the `StrReverse()` function reverses the order of the characters in the `userName` string and the `&` (ampersand) character is used to concatenate strings and rearrange the user's name such that the last name is first, followed by a comma and the first name.

HINT

String concatenation is the process of combining one or more strings together to form a new string. The strings are combined from left to right using either the ampersand (&) or addition (+) operators. To avoid ambiguity with the mathematical addition operator, I recommend that you always use the ampersand (&) operator for string concatenation.

1.4 Procedures and Conditions

Although the two topics in this Section title don't necessarily go hand in hand, they do represent basic constructs essential for any program. In this Section, you closely observe both procedures and conditions in order to establish some basic tools with which to work in VBA.

Specifically, in this Section I will discuss:

- Sub Procedures
- Function Procedures
- Event Procedures
- Conditional Logic

- Conditional Statements and the If/Then/Else and Select/Case Code Structures

1.4.1 VBA Procedures

You may remember that a module is a segment of your project that contains a related set of declarations and procedures. You may also remember that every module has its own window within the VBA IDE and, depending on whether or not it is an object module or a standard module, slightly different behavior regarding variables. Programming procedures can be constructed within each of these module windows if they are not already defined. Let's take a look at the different type of procedures that can be used and/or built using VBA.

1.4.1.1 Event Procedures

You have already seen a few examples of event procedures; such as the Click() event procedure of a Command Button control, and the SelectionChange() event procedure of a worksheet. VBA predefines these procedures in the sense that you cannot change the name of the procedure, nor the object within Excel to which the procedure belongs, nor the conditions under which the procedure is triggered. For the most part, all you can do with these procedures is add the code to be executed when the event is triggered. Typically, several events are associated with each Excel object; whether it is a worksheet, workbook, chart, or ActiveX control. Figure 1.19 shows the object module for a worksheet and displays all of the events associated with a worksheet in Excel.

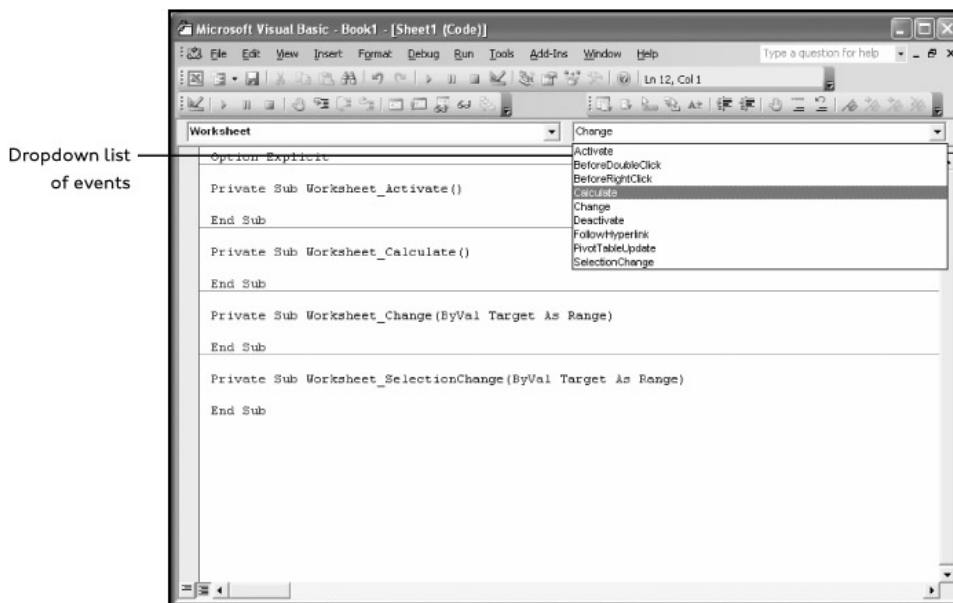


Figure 1.19 - Worksheet events in Excel.

Event procedures are defined with the Sub keyword followed by the name of the procedure.

```
Private Sub Worksheet_Activate()  
    'Event procedure code is listed here.  
End Sub
```

The name of the procedure listed above is Worksheet_Activate(), although it will be more commonly referred to as the Activate() event. No parameters are passed to this procedure because the parentheses are empty. This procedure is triggered when the worksheet to which it is associated is activated; that is, when you switch between two different windows or worksheets, the Activate() event of the currently selected worksheet is triggered. The procedure ends with the line End Sub, unless the statement Exit Sub is used within the procedure code.

1.4.1.2 Parameters with Event Procedures

Parameters are the list of one or more variables passed to the event procedure when it is triggered. The values of the parameters passed to the event procedure contain information related to the event. A comma separates multiple variables, and the variable data type is also declared. VBA defines everything about the parameters passed to the event procedure; including the number of parameters, the name of each parameter and their data types, and the method in which they are passed. Although it is possible to change the name of the variables in the parameter list under certain circumstances, I do not recommend editing the event procedure definition in any way.

The following example shows the MouseDown() event procedure of a Command Button control. This procedure triggers when the user clicks on the Command Button control with the mouse. The first and last lines of the procedure are automatically created by VBA. I added the four lines of code within the procedure.

```
Private Sub CommandButton1_MouseDown(ByVal Button As Integer, ByVal Shift
As Integer,
ByVal X As Single, ByVal Y As Single)
    Range("A2").Value = Button
    Range("B2").Value = Shift
    Range("C2").Value = X
    Range("D2").Value = Y
End Sub
```

There are four parameters passed to the MouseDown() event procedure: Button, Shift, X, and Y; they have all been declared as numerical data types. These parameters contain numerical information describing the event that just occurred, and they can be used as variables within the procedure because they have already been declared. The ByVal keyword will be discussed later in this chapter, so just ignore it for now. The previous code was added to the MouseDown() event procedure of a Command Button control placed on a worksheet with a few column headers as shown in Figure 1.20.

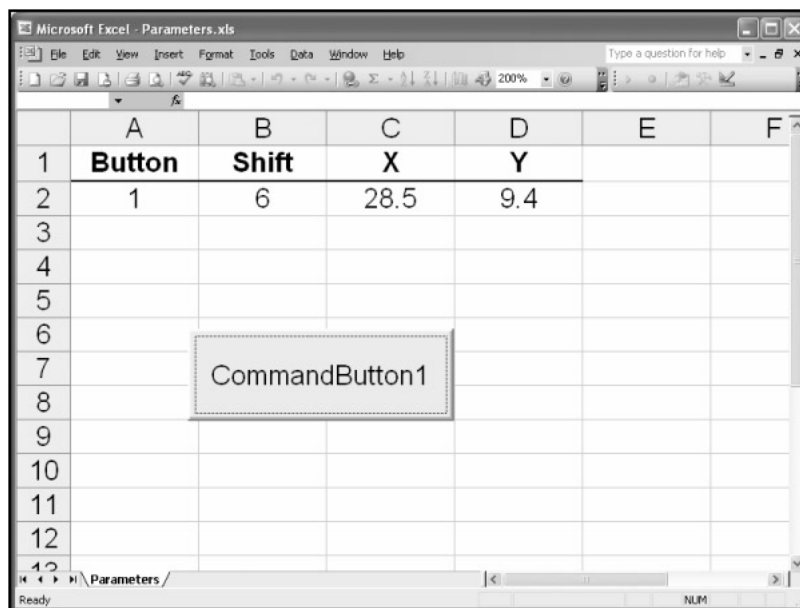


Figure 1.20 - Parameter values of the MouseDown() event procedure.

The values of the parameter variables are copied to the appropriate cells in this worksheet when the user clicks on the Command Button control. The variable Button represents the mouse button that was clicked—a value of 1 for the left mouse button, 2 for the right mouse button, and 3 for the middle mouse button. The variable Shift represents whether the Shift key was pressed (1) or not (0). The variables X and Y represent the horizontal and vertical coordinates of the click in units of the worksheet's width and height, respectively.

middle mouse button (if it exists). The variable Shift represents the combination of Shift, Ctrl, and Alt keys held down while the mouse button was clicked. Since there are eight possible combinations of these three keys, the variable Shift can hold an integer value between zero and seven. The variables X and Y represent the location of the mouse cursor within the Command Button control when the mouse button was clicked. The values of X and Y fall within zero to the value of the Width property of the Command Button control for X, and zero to the value of the Height property for Y. The upper left corner of the Command Button control is X = 0, Y = 0.

You now see how helpful the information within these parameters can be. For example, a programmer might use the MouseDown() and MouseUp() event procedures of an ActiveX control to catch a right click of the mouse button on the control. The MouseDown() event procedure might be used to display a menu with various options, and the MouseUp() event procedure would then be used to hide the menu. Does this sound familiar?

It is both impractical and unnecessary to discuss all of the event procedures of all Excel objects and ActiveX controls in this book. The examples you have seen so far are a good representation of how to use event procedures in VBA. In order to establish which event procedures (if any) should be used in your program, do the following:

- Ask yourself, "When should something happen?"
- Search for the event procedure(s) that will be triggered by the answer to the question, "When should something happen?" The event procedures have sensible names related to the action that triggers them; however, it may be useful to look up the description of the event procedure in the online help.
- If you cannot find an event procedure that triggers when desired, redesign your program with ActiveX controls that do contain a useful event procedure. If you still can't find anything, then there are probably errors in the logic of your algorithm.
- Test possible procedures by writing simple programs such as the one for the MouseDown() event procedure listed earlier.
- Insert the code that carries out the tasks you want once you recognize the proper event procedure.

1.4.1.3 Private, Public, and Procedure Scope

The Private and Public keywords used with procedure definitions have a similar function to that used with variable declarations. Private and Public are used to define the procedure's scope. The Public keyword makes the procedure visible to all other procedures in all modules in the project. The Private keyword ensures that the procedure is visible to other procedures within the same module, but keeps it inaccessible to all other procedures outside the module in which it is defined. The Private and Public keywords are optional, but VBA includes them in predefined event procedures. If Private or Public is omitted, then the procedure is public by default.

TRICK

Use the Option Private statement in the general declarations section of a module to keep public modules visible only within the project. Omit Option Private if you wish to create reusable procedures that will be available for any project.

1.4.1.4 Sub Procedures

Although all procedures are really sub (short for *subroutine*) procedures, I will use the term to refer to those procedures created entirely by the programmer. The basic syntax and operation of a sub

procedure is the same as for an event procedure. You define the procedure with the scope using the Public or Private keywords, followed by the keyword Sub, the procedure name, and the parameter list (if any). Sub procedures end with the End Sub statement. You can either type in the procedure definition or use the Insert/Procedure menu item to bring up the Add Procedure dialog box, as shown in

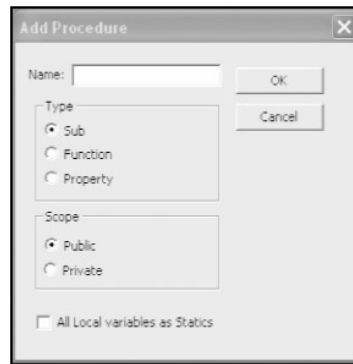


Figure 1.21.

```
Private Sub myProcedure(parameter list)
    'Sub procedure code is listed here.
End Sub
```

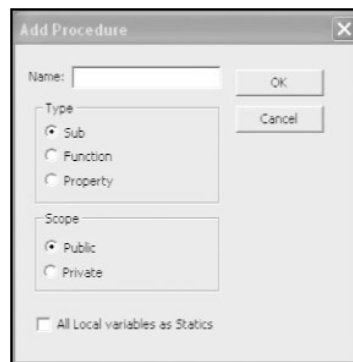


Figure 1.21-The Add dialog box.

The MsgBox() function is a good way to alert the user about some type of problem, or ask a question that requires a yes/no answer. Here is the syntax for the MsgBox() function:

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

The prompt is the only required parameter, although buttons and title are usually included. The example below was used to generate the message box in **Error! Reference source not found.:**

```
userResponse = MsgBox("Testing the Message Box", vbOKOnly, "Message")
```

The prompt must be a string or string variable and is used as the message you want the user to read. The buttons parameter requires a numeric expression (either an integer or constant) and tells VBA what buttons and/or icons are to be placed on the message box. There are several choices for buttons, including OK, OK/Cancel, Abort/Retry/Ignore, and Yes/No. You can also display an icon (warnings or information type), a help button, and add some additional formatting with your choice

of buttons. For a complete list of button choices, look up the MsgBox() function in the on-line help by typing **msgbox** in the keyword field of the help window.

Finally, you should take care not to use too many message boxes in your program. Always ask yourself: are there other ways to get input or display the message besides including a message box? Most users (including myself) find it extremely annoying to have to answer a message box when it's not really necessary.

1.4.2 Manipulating Strings with VBA Functions

Now it's time to get back to strings and have a little fun. Strings are more of an unknown to the programmer in the sense that you seldom know how long they are, or how much of the string actually represents useful information. Thankfully, there is a plethora of functions designed to work on string variables that you can use to extract the information you need. Table 1.3 summarizes many of these functions.

As with most functions, the string functions require one or more parameters be passed. All functions must return a value so the syntax will look something like this:

```
myVar = FunctionName(parameter list)
```

where myVar is a variable of the proper type for the return value of the function, FunctionName is the name of the VBA function, and parameter list is a list of one or more values to be passed to the function. Parameters can be literals (for example, 5.2 or "Hello"), but are usually in the form of variables.

Table 1.3 - VBA string functions

Function Name	Returns
Str()	A string representation of a number
Val()	A numerical representation of a string
Trim()	A string with leading and trailing spaces removed
Left()	A portion of a string beginning from the left side
Right()	A portion of a string beginning from the right side
Mid()	Any portion of a string
InStr()	A number representing the place value of a particular character within a string
InStrRev()	The position of an occurrence of one string within another, from the end of string
StrReverse()	A string with its character order reversed
Len()	A number of characters in a string
LCase()	A string with all characters lowercase
UCase()	A string with all characters uppercase
StrConv()	A string converted to one of several possible formats
StrComp()	A number indicating the result of a string comparison
Asc()	Number representing the ANSI code of a character

1.4.2.1 Fun with Strings

The best way to learn these functions is to use them, so let's create a program that asks for the user's name and then outputs components of the name to a worksheet. I call it *Fun with Strings*, and Figure 1.18 shows the spreadsheet.

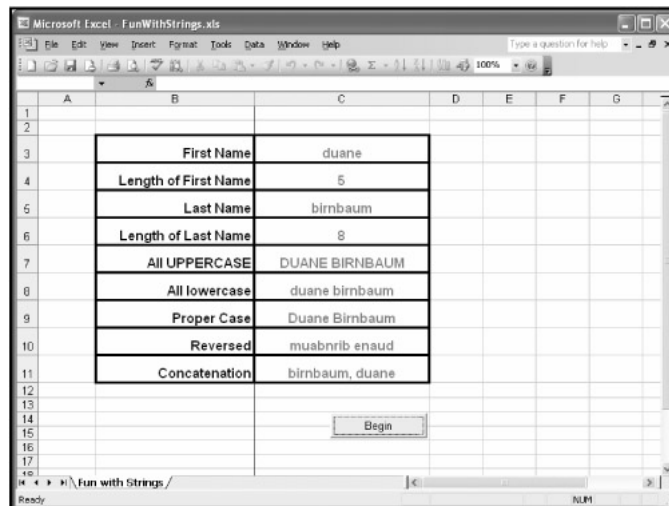


Figure 1.18 - Fun with Strings.

Specifically, the program will output the user's first name and last name along with the number of characters in each name to separate cells in the spreadsheet. The program will also convert the user's name to both all uppercase and all lowercase characters as well as reverse the order of the first and last name. The code is placed in the Click() event procedure of a Command Button control placed on the worksheet. The Name property of the Command Button control was changed to cmdBegin and the Caption property to "Begin". When the user clicks on the command button, code execution begins. After some variable declarations, the InputBox() function is used to prompt the user for his/her first and last name. You will notice that I am assuming the user enters his/her first name followed by one space and then the last name.

Everything entered by the user is stored in the string variable userName.

```
Private Sub cmdBegin_Click()
    Dim userName As String
    Dim firstName As String
    Dim lastName As String
    Dim strLength As Integer
    Dim spaceLoc As Integer

    '-----
    'Collect user name, find the space between
    'first and last names, and separate the names.
    '-----
    userName = InputBox("Enter your first and last name.", "Name")
    spaceLoc = InStr(1, userName, " ")
    firstName = Left(userName, spaceLoc - 1)

    '-----
    'Output to the worksheet
    '-----
    Range("C3").Value = firstName
    strLength = Len(firstName)
    Range("C4").Value = strLength      'length of first name
    strLength = Len(userName)
    lastName = Mid(userName, spaceLoc + 1, strLength - spaceLoc)
    Range("C5").Value = lastName
    strLength = Len(lastName)
    Range("C6").Value = strLength
    Range("C7").Value = UCase(userName)
    Range("C8").Value = LCase(userName)
```

```

Range("C9").Value = StrConv(userName, vbProperCase)
Range("C10").Value = StrReverse(userName)
Range("C11").Value = lastName & ", "& firstName
End Sub

```

To help picture what will happen in the program, let's assume the variable `userName` contains the string "Fred Flintstone". This string is 15 characters long; Table 1.4 shows the locations of each character.

Table 1.4- Character locations in a string

Character	F	r	e	d		F	l	i	n	t	s	t	o	n	e
Location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The program determines the location of the space by using the `InStr()` function. The `InStr()` function is passed three parameters, the number 1, the string variable `userName`, and a single character string containing a space. The parameter 1 represents the location to start searching within the string passed in the next parameter, in this case, `userName`. The last string is a space and this represents the character the `InStr()` function is searching for within the value of `userName`. The `InStr()` function then returns an integer value representing the location of the space within the `userName` string. This integer value is the location of the space between the first and last name of the user—in this example, location 5 (see Table 1.4)—and is stored in the integer variable `spaceLoc`. The `Left()` function is then passed two parameters, the `userName` string, and the length of the portion of the `userName` string to return. The variable `spaceLoc` is holding the location of the space (5 in our example), so using `spaceLoc - 1` for the length parameter in the `Left()` function returns just the first name ("Fred"). The `Len()` function is used to return the length of the `firstName` string as an integer and this value is stored in the variable `strLength`. The values of the `firstName` string and `strLength` variables are then copied to the worksheet.

The `Mid()` function is used to return the last name of the user to the string variable `lastName`. The `Mid()` function takes three parameters: the original string `userName` ("Fred Flintstone"), the starting location of the new string (`spaceLoc - 1`), and the length of the string to return (`strLength - spaceLoc`). The variable `strLength` was reinitialized to the length of `userName` prior to using the `Mid()` function. Again, the variables holding the last name and the number of characters in the last name are copied to the worksheet.

The `UCase()` and `LCase()` functions convert the `userName` string to all uppercase and all lowercase letters, respectively; and the `StrConv()` function converts the `userName` string to proper case. Finally, the `StrReverse()` function reverses the order of the characters in the `userName` string and the `&` (ampersand) character is used to concatenate strings and rearrange the user's name such that the last name is first, followed by a comma and the first name.

HINT

String concatenation is the process of combining one or more strings together to form a new string. The strings are combined from left to right using either the ampersand (&) or addition (+) operators. To avoid ambiguity with the mathematical addition operator, I recommend that you always use the ampersand (&) operator for string concatenation.

1.5 Procedures and Conditions

Although the two topics in this Section title don't necessarily go hand in hand, they do represent basic constructs essential for any program. In this Section, you closely observe both procedures and conditions in order to establish some basic tools with which to work in VBA.

Specifically, in this Section I will discuss:

- Sub Procedures
- Function Procedures
- Event Procedures
- Conditional Logic
- Conditional Statements and the If/Then/Else and Select/Case Code Structures

1.5.1 VBA Procedures

You may remember that a module is a segment of your project that contains a related set of declarations and procedures. You may also remember that every module has its own window within the VBA IDE and, depending on whether or not it is an object module or a standard module, slightly different behavior regarding variables. Programming procedures can be constructed within each of these module windows if they are not already defined. Let's take a look at the different type of procedures that can be used and/or built using VBA.

1.5.1.1 Event Procedures

You have already seen a few examples of event procedures; such as the Click() event procedure of a Command Button control, and the SelectionChange() event procedure of a worksheet. VBA predefines these procedures in the sense that you cannot change the name of the procedure, nor the object within Excel to which the procedure belongs, nor the conditions under which the procedure is triggered. For the most part, all you can do with these procedures is add the code to be executed when the event is triggered. Typically, several events are associated with each Excel object; whether it is a worksheet, workbook, chart, or ActiveX control. Figure 1.19 shows the object module for a worksheet and displays all of the events associated with a worksheet in Excel.

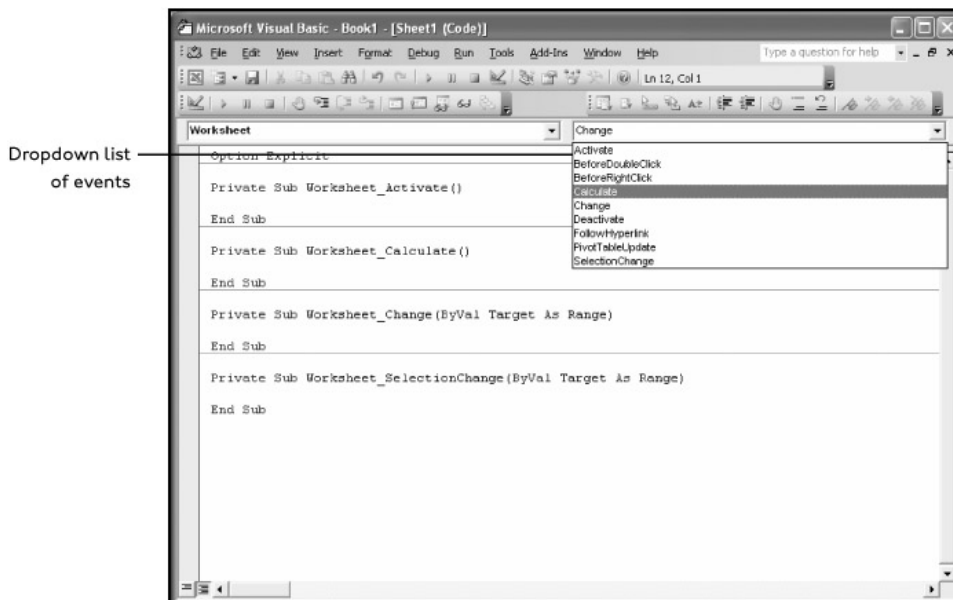


Figure 1.19 - Worksheet events in Excel.

Event procedures are defined with the Sub keyword followed by the name of the procedure.

```
Private Sub Worksheet_Activate()  
    'Event procedure code is listed here.  
End Sub
```

The name of the procedure listed above is `Worksheet_Activate()`, although it will be more commonly referred to as the `Activate()` event. No parameters are passed to this procedure because the parentheses are empty. This procedure is triggered when the worksheet to which it is associated is activated; that is, when you switch between two different windows or worksheets, the `Activate()` event of the currently selected worksheet is triggered. The procedure ends with the line `End Sub`, unless the statement `Exit Sub` is used within the procedure code.

1.5.1.2 Parameters with Event Procedures

Parameters are the list of one or more variables passed to the event procedure when it is triggered. The values of the parameters passed to the event procedure contain information related to the event. A comma separates multiple variables, and the variable data type is also declared. VBA defines everything about the parameters passed to the event procedure; including the number of parameters, the name of each parameter and their data types, and the method in which they are passed. Although it is possible to change the name of the variables in the parameter list under certain circumstances, I do not recommend editing the event procedure definition in any way.

The following example shows the `MouseDown()` event procedure of a Command Button control. This procedure triggers when the user clicks on the Command Button control with the mouse. The first and last lines of the procedure are automatically created by VBA. I added the four lines of code within the procedure.

```
Private Sub CommandButton1_MouseDown(ByVal Button As Integer, ByVal Shift
As Integer,
ByVal X As Single, ByVal Y As Single)
    Range("A2").Value = Button
    Range("B2").Value = Shift
    Range("C2").Value = X
    Range("D2").Value = Y
End Sub
```

There are four parameters passed to the `MouseDown()` event procedure: `Button`, `Shift`, `X`, and `Y`; they have all been declared as numerical data types. These parameters contain numerical information describing the event that just occurred, and they can be used as variables within the procedure because they have already been declared. The `ByVal` keyword will be discussed later in this chapter, so just ignore it for now. The previous code was added to the `MouseDown()` event procedure of a Command Button control placed on a worksheet with a few column headers as shown in Figure 1.20.

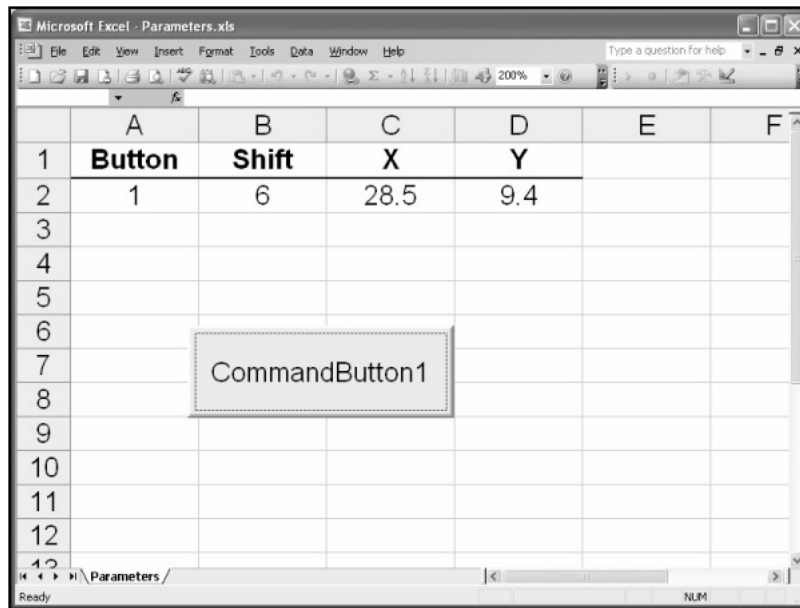


Figure 1.20 - Parameter values of the MouseDown() event procedure.

The values of the parameter variables are copied to the appropriate cells in this worksheet when the user clicks on the Command Button control. The variable **Button** represents the mouse button that was clicked—a value of 1 for the left mouse button, 2 for the right mouse button, and 3 for the middle mouse button (if it exists). The variable **Shift** represents the combination of Shift, Ctrl, and Alt keys held down while the mouse button was clicked. Since there are eight possible combinations of these three keys, the variable **Shift** can hold an integer value between zero and seven. The variables **X** and **Y** represent the location of the mouse cursor within the Command Button control when the mouse button was clicked. The values of **X** and **Y** fall within zero to the value of the Width property of the Command Button control for **X**, and zero to the value of the Height property for **Y**. The upper left corner of the Command Button control is **X** = 0, **Y** = 0.

You now see how helpful the information within these parameters can be. For example, a programmer might use the **MouseDown()** and **MouseUp()** event procedures of an ActiveX control to catch a right click of the mouse button on the control. The **MouseDown()** event procedure might be used to display a menu with various options, and the **MouseUp()** event procedure would then be used to hide the menu. Does this sound familiar?

It is both impractical and unnecessary to discuss all of the event procedures of all Excel objects and ActiveX controls in this book. The examples you have seen so far are a good representation of how to use event procedures in VBA. In order to establish which event procedures (if any) should be used in your program, do the following:

- Ask yourself, "When should something happen?"
- Search for the event procedure(s) that will be triggered by the answer to the question, "When should something happen?" The event procedures have sensible names related to the action that triggers them; however, it may be useful to look up the description of the event procedure in the online help.
- If you cannot find an event procedure that triggers when desired, redesign your program with ActiveX controls that do contain a useful event procedure. If you still can't find anything, then there are probably errors in the logic of your algorithm.
- Test possible procedures by writing simple programs such as the one for the **MouseDown()** event procedure listed earlier.

- Insert the code that carries out the tasks you want once you recognize the proper event procedure.

1.5.1.3 Private, Public, and Procedure Scope

The Private and Public keywords used with procedure definitions have a similar function to that used with variable declarations. Private and Public are used to define the procedure's scope. The Public keyword makes the procedure visible to all other procedures in all modules in the project. The Private keyword ensures that the procedure is visible to other procedures within the same module, but keeps it inaccessible to all other procedures outside the module in which it is defined. The Private and Public keywords are optional, but VBA includes them in predefined event procedures. If Private or Public is omitted, then the procedure is public by default.

TRICK

Use the Option Private statement in the general declarations section of a module to keep public modules visible only within the project. Omit Option Private if you wish to create reusable procedures that will be available for any project.

1.5.1.4 Sub Procedures

Although all procedures are really sub (short for *subroutine*) procedures, I will use the term to refer to those procedures created entirely by the programmer. The basic syntax and operation of a sub procedure is the same as for an event procedure. You define the procedure with the scope using the Public or Private keywords, followed by the keyword Sub, the procedure name, and the parameter list (if any). Sub procedures end with the End Sub statement. You can either type in the procedure definition or use the Insert/Procedure menu item to bring up the Add Procedure dialog box, as shown in

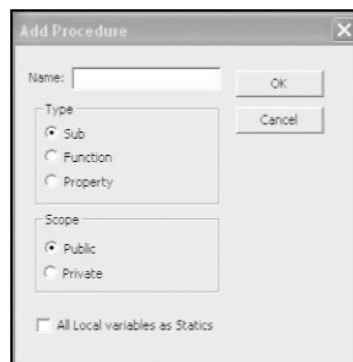


Figure 1.21.

```
Private Sub myProcedure(parameter list)
    'Sub procedure code is listed here.
End Sub
```

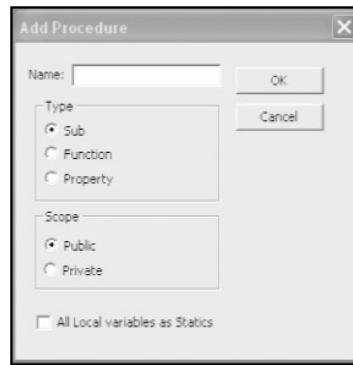


Figure 1.21-The Add dialog box.

Sub procedures differ from event procedures in that:

- the programmer defines the procedure name and any variable names in the parameter list.
- the programmer decides how many (if any) variables are in the parameter list.
- they can be placed in both object and standard modules.
- execution begins when they are "called" using code from other parts of the program and cannot be automatically triggered.

The following program collects two numbers from the user, adds them, and outputs the result. This program can reside in any module. For simplicity, I tested this program by running it directly from the VBA IDE. To begin program execution from the VBA IDE, first insert the mouse cursor within the procedure to be executed, and then press F5 or select the appropriate icon from the Standard toolbar or Run menu, as shown in Figure 1.22.

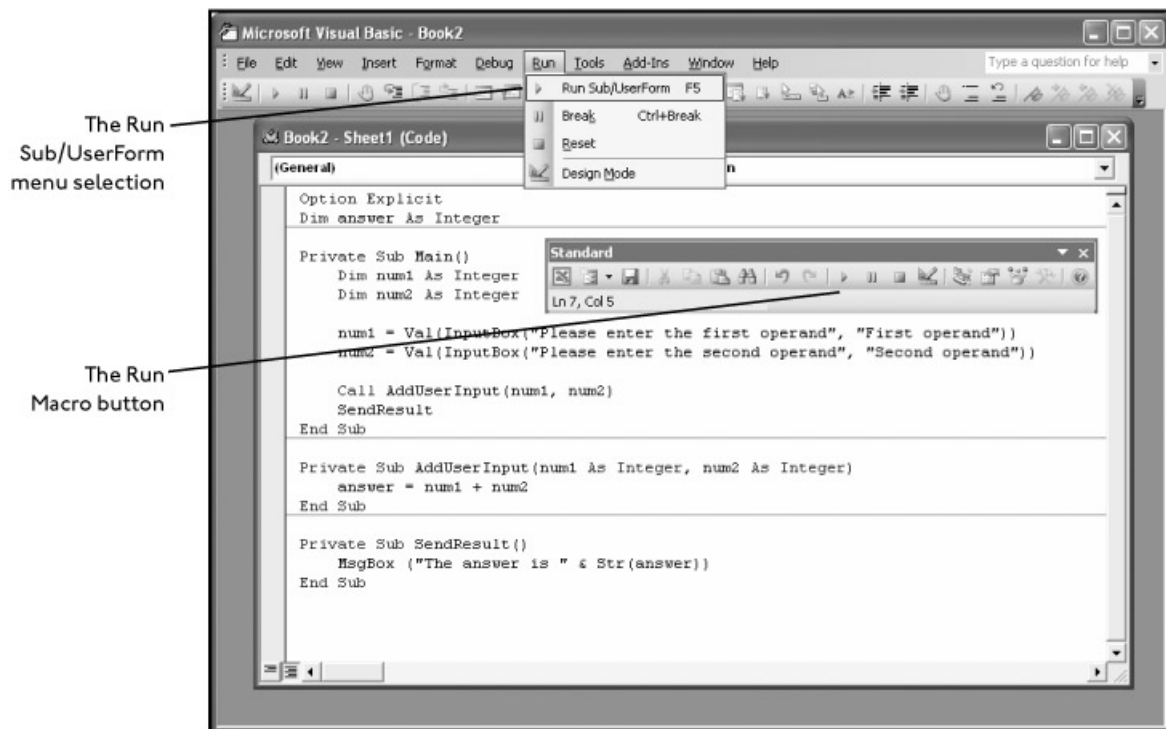


Figure 1.22 - Running a program from the VBA IDE.

```
Option Explicit
Dim answer As Integer

Private Sub Main()
    Dim num1 As Integer
    Dim num2 As Integer
    num1 = Val(InputBox("Please enter the first operand", "First operand"))
    num2 = Val(InputBox("Please enter the second operand", "Second operand"))

    Call AddUserInput(num1, num2)
    SendResult
End Sub

Private Sub AddUserInput(num1 As Integer, num2 As Integer)
    answer = num1 + num2
End Sub

Private Sub SendResult()
    MsgBox ("The answer is "& Str(answer))
End Sub
```

First, variable declaration is required with Option Explicit and a module level variable (answer) is declared. The majority of the program is listed in the sub procedure Main(). The sub procedure Main() is declared as Private and serves as the central procedure for the program. Two procedure-level integer variables (num1 and num2) are declared and assigned to the return value of input boxes. The Val() function is used to convert the string type return value from the InputBox() function to a numerical value.

After two values are input by the user, the program makes the calls to the sub procedures AddUserInput() and SendResult(). The Call keyword is used to send program execution to AddUserInput() and the variables num1 and num2 are passed to this procedure. The Call keyword is required when passing parameters enclosed in parentheses; otherwise it is unnecessary (for example, AddUserInput num1, num2 is an identical statement). After the AddUserInput() procedure executes, program execution resumes in the Main() procedure where it left off. The line SendResult is another procedure call and sends program execution to the SendResult() sub procedure. As no parameters are passed, the Call keyword is omitted (although you may include it if you like). The Main() procedure, and consequently the program, terminates after program execution returns from the SendResult() procedure. The AddUserInput() procedure's only purpose is to accept the two addends from the Main() procedure, add them together, and store the result in the module level variable answer. Note that I used the same variable names for the two addends when defining the AddUserInput() procedure. This is perfectly legitimate, as this is outside the scope of the original num1 and num2 variables. Finally, the SendResult() procedure is used to output the answer using a basic message box. A Str() function is used to convert the numerical variable answer to a string before it is concatenated to the rest of the message.

TRICK

Keep your procedures as short as possible. You will find that as your procedures get longer, they get harder to read and debug. As a general rule I try to keep my procedures to a length such that all of the code is visible on my monitor. If your procedure gets much longer than one screen, break the procedure into two or more procedures.

1.5.1.5 ByVal and ByRef

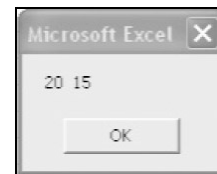
You should have noticed the ByVal keyword in the parameter list of theMouseDown() event procedure shown earlier in the chapter. The ByVal keyword tells VBA to make a copy of the value stored in the accompanying variable. Thus, any manipulation of the copied value within the procedure does not affect the original variable.

The alternative to passing a variable by value is to pass a variable to another procedure by reference; the ByRef keyword is used to do so. When you pass by reference you are essentially passing the original variable to the procedure. Any manipulation of the variable in the new procedure is permanent, so the variable does not retain its original value when program execution proceeds back to the calling procedure.

This is true even if you use a new variable name in the procedure that accepts the variable passed by reference. Passing by reference is the default behavior, so you can omit the ByRef keyword if you wish.

The following short program will make the behavior of ByVal and ByRef clear. I suggest inserting a new module into a project, adding the code below, and running the program from the procedure Main().

```
Private Sub Main()
    Dim num1 As Integer
    Dim num2 As Integer
    num1 = 10
    num2 = 15
    Call PassByRef(num1)
    Call PassByVal(num2)
    MsgBox (num1 & " " & num2)
End Sub
Private Sub PassByRef(ByRef num3 As Integer)
    num3 = 20
End Sub
Private Sub PassByVal(ByVal num2 As Integer)
    num2 = 20
End Sub
```



First two integer variables are declared and initialized to the values 10 and 15. The first variable, num1, is passed by reference to the procedure PassByRef() in a variable called num3. The value 20 is assigned to the num3 variable inside the PassByRef() procedure. Next the variable num2 is passed by value to the PassByVal() procedure, where it is copied to another variable called num2. The num2 variable in the PassByVal() procedure is then assigned the value 20. The program ends with the output of the original num1 and num2 variables in a message box.

Now ask yourself: "What values output in the message box?" The answer is 20 for the num1 variable, and 15 for the num2 variable. The variable num1 holds the value 20 at the end of the Main() procedure because it was changed in the PassByRef() procedure. Even though a different variable name was used in the PassByRef() procedure, the num3 variable still refers to the same memory location holding the value assigned to the num1 variable. Essentially, we have one variable with two names, each with its own scope. The num2 variable retains its value of 15 at the end of Main() procedure because it had been passed by value to the Pass-ByVal() procedure. Passing by value makes a copy of the variable's value to a new variable, even if the variable in the accepting procedure (PassByVal) has the same name. In this case, there are two variables with the same name.

You pass a variable by reference to a procedure in order to change the value of the original variable; or when the variable is needed in the procedure, but its value does not have to be changed. If the variable needs to be altered for another purpose but must retain its original value; then pass the variable by value using the ByVal keyword.

1.5.1.6 Function Procedures

Function procedures are very much like other procedures with one significant difference: they return a value to the calling procedure. Now you might be concerned or confused by the fact that I used the term *functions* before in reference to Excel's spreadsheet functions and VBA's string and date functions. So, what's the difference between these two terminologies? There is no difference. Everything I have, or will call a function is essentially the same thing. A *function* is a small program built with a specific purpose that, when used, will return a value to the calling procedure or spreadsheet cell(s).

If you are familiar with the built-in functions available in the Excel application, such as SUM(), AVERAGE(), STDEV(), then you already have a basic understanding of how they work. Functions are often (but not always) passed one or more values and they always return at least one value. For example, if I enter the formula =AVERAGE(A2:A10) into cell A11 on a work-sheet in the Excel application, I know that the average of the nine values given in the range A2:A10 will be calculated and returned to cell A11. Excel recognizes the AVERAGE keyword in the formula as one of its built-in functions. Excel then calls the function procedure AVERAGE() and passes the range of values specified in parentheses—in this case, 9 values. The function procedure AVERAGE() then calculates the average of the values passed in as parameters and returns the result to the spreadsheet cell containing the formula. In VBA, you can also call function procedures such as Left(), Mid(), and DateDiff(), as you have seen in previous examples. You can even use the built-in functions of the Excel application. Finally, you can create your own function procedures in VBA.

1.5.1.7 Creating Your Own VBA Functions

The basic syntax for creating a function procedure in VBA is as follows:

```
Private/Public Function FunctionName(parameter list) as type
    'Function procedure code is listed here
    FunctionName = Return value
End Function
```

This is similar to the syntax for any procedure with the procedure name, parameter list, and an End statement. You can, and should include a Private or Public keyword to define the scope of the function. One obvious difference is the Function keyword replaces Sub. Also, you should define a return type to the function. The return data type is used for the value that the function sends back to the calling procedure. If you do not specify the data type, then the function's return value will be of type variant. The function returns a value by assigning the desired value to the name of the function, although the return value is usually stored in a variable.

TRICK

Use Exit Sub or Exit Function if you need to return program execution to the calling procedure before the rest of the code in the procedure executes.

```
myVar = MyFunction(param1)
```

Here, the variable myVar is assigned the return value of the function named MyFunction() that is passed one parameter in the form of a variable named param1.

Now let's consider an example of a function that mimics one of Excel's built-in functions. The following function calculates the result of raising a number to a specified power. I named the function PowerDB() and set its return value as type double. The PowerDB() function accepts two numerical values for input, the number to which the exponent will be applied (number), and the value of the exponent (n). The function has been given public scope.

The code is really very simple. The value of the variable number is raised to the power of the value of the variable n, and then the result is restored in the variable number. The value of the variable number is assigned to the function so that it may be returned to the calling procedure.

```
Public Function PowerDB(ByVal number As Double, n As Single) As Double
    number = number ^ n
    PowerDB = number
End Function
```

A procedure that utilizes the PowerDB() function can be written as follows:

```
Private Sub TestPower()
    Dim number As Double
    Dim n As Single
    Dim result As Double
    number = Val(InputBox("Enter a number.", "Number"))
    n = Val(InputBox("Enter the value of the exponent.", "Exponent"))
    result = PowerDB(number, n)
    MsgBox (number & "^" & n & " = " & result)
End Sub
```

The only new idea here is the line that calls the PowerDB() function, result = PowerDB(number, n). The variable result is assigned the return value of the function and output in a message box. Note that the data types for the PowerDB() function and variable result match (double). The variable number was passed to the PowerDB() function by value because if I passed it by reference its value would be changed by the function. Since I want to use the original value of number in the final output, I must pass it by value. The variable n was passed by reference because I did not change its value in the function procedure and VBA is more efficient when passing values by reference.

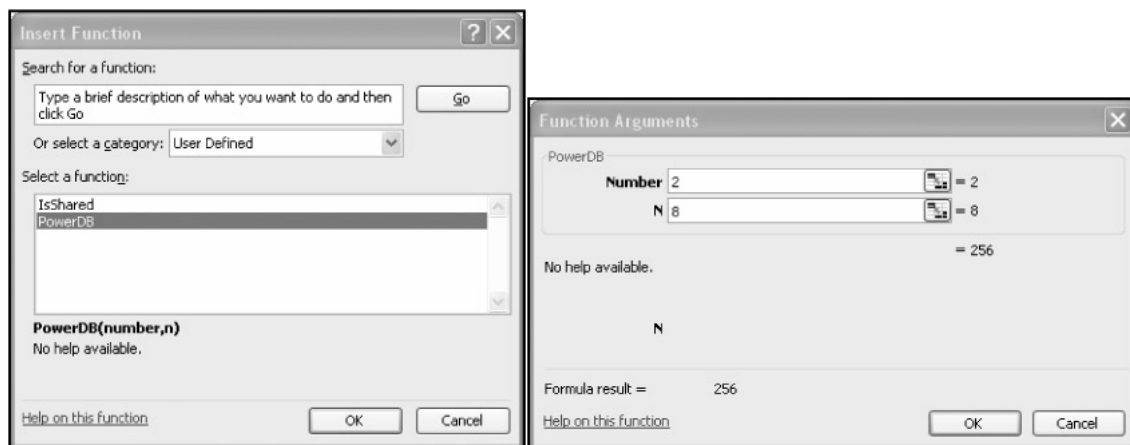


Figure 1.23 – Insert function tool, step 1 and 2

A public scope for the function PowerDB() makes it visible to all procedures in the project and the Excel application provided the function is contained in a standard module. Thus, this function can

now be used like any other function in Excel. Returning to the Excel application and entering the formula =PowerDB(2,8) into any worksheet cell will return the value 256 to that cell. The PowerDB() function is even listed in Excel's insert function tool as shown in Figure 1.23.

You now see that I named the function PowerDB() in order to avoid a conflict with Excel's POWER() function. You can create your own library of VBA functions to use in your spread-sheet applications. Keeping a library of VBA functions saves you valuable time as you do not have to re-write these functions to use them in another project.

1.5.1.8 Using Excel Application Functions in VBA

Now that you know how to write functions in VBA and make them available to your spread-sheets, you are also aware that you can re-create any function already available in the Excel application. Although recreating Excel's functions would be a good way to improve your VBA programming skills, it's certainly not a practical use of your time. Why reinvent what's already been created for you? It would be nice if you could use Excel's application functions in your VBA code, as they are mostly complimentary, not repetitive, to VBA's set of functions. That way, if you need a specific function performed in your program that is not already included with VBA, you don't have to write it yourself.

Well, there *is* a method to use the Excel application functions, of course, and it is really quite simple.

```
result = Application.WorksheetFunction.Power(number, n)
```

Replacing the call to the PowerDB() function in the TestPower() sub procedure shown earlier with the line of code above will give the exact same result. The difference is that this code uses Excel's POWER() function and not the PowerDB() function. You can probably guess what's happening from the names used in this line of code. The component Application.WorksheetFunction will return all functions available from the Excel application. From there it is a simple matter of adding on the name of the function and inserting the required parameters into the parentheses. Two more examples illustrate the use of the AVERAGE() and STDEV() functions from the Excel application.

```
myVar = Application.WorksheetFunction.Average(5, 7, 9)
myVar2 = Application.WorksheetFunction.StDev(3, 7, 11)
```

The examples above will return the value 7 to the variable myVar and 4 to the variable myVar2.

1.5.1.9 Logical Operators with VBA

- Logic as applied to a computer program is evaluating an expression as true or false. An expression is typically, but not always, a comparison of two variables such as var1>var2 or var1=var2 (see Table 1.5 for a list of available comparison operators).

Table 1.5 - Comparison operators in VBA

Operator	Function
=	Tests for equality
<>	Tests for inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

A programmer reads these expressions as follows:

- The value of var1 is greater than the value of var2.
- The value of var1 equals the value of var2.

The statements are evaluated as true or false.

Imagine a simple device that takes a single expression as input, evaluates that expression as true or false, spits out the answer, and then moves on to the next expression. The evaluation of the expression is a simple task since there are only two choices and computers are very good at assigning 1's (true) or 0's (false) to things. The difficulty arises from trying to make sense out of the expressions that have been evaluated as true or false. This is where Boolean (after the nineteenth century mathematician George Boole) algebra comes in to play. Boolean algebra refers to the use of the operators AND, OR, NOT, and a few others to evaluate one or more expressions as true or false. Then, based on the result of the logic, the program selects a direction in which to proceed.

AND, OR, and NOT Operators

VBA uses logical AND to make a decision based on the value of two conditions. The value of each condition can be one of two values, true or false. Consider the following two conditions.

Condition 1	Condition2
myVar > 10	myVar < 20

The expression Condition1 AND Condition2 evaluates as true only if Condition1 and Condition2 are both true. If either or both conditions evaluate to false then the overall result is false. The evaluation of expressions using logical operators is easily displayed in truth tables. Table 1.6 shows the truth table for logical AND.

Table 1.6 - Truth table for the AND operator

Condition1	Condition2	Condition1 AND Condition2
True	True	True
True	False	False
False	True	False
False	False	False

The logical operator OR returns true from an expression when at least one of the conditions within the expression is true. The expression Condition1 OR Condition2 evaluates as true when either Condition1 or Condition2 is true or if both conditions are true. Table 1.7 shows the truth table for logical OR.

Table 1.7 - Truth table for the OR operator

Condition1	Condition2	Condition1 OR Condition2
True	True	True
True	False	True
False	True	True
False	False	False

1.5.1.10 Conditionals and Branching

It may seem like I've covered a fair amount of VBA programming, but in reality, I've barely scratched the surface. Right now, you can't really do much with the VBA programs you've written,

because you haven't learned any programming structures; however, that is about to change as I begin to examine a simple yet very useful VBA code structure. The If/Then/Else structure is known as both a conditional and branching structure because it uses conditional statements to change the flow or direction of program execution.

If/Then/Else

There are several ways to implement this code structure. The most basic uses the two required keywords If and Then.

```
If (condition) Then Code statement
```

In the example above, the code statement following *Then* will execute if condition evaluates as true; otherwise code execution proceeds with the next statement. The entire structure takes just one line of code. It's convenient when you have just one brief code statement that needs to be executed if the condition is true. Multiple statements can be entered on the same line if you separate them with colons (:), but then your code may be hard to read. If you need more than one code statement executed, then for the sake of readability, you should use the block form of If/Then.

```
If (condition) Then
    'Block of code statements
End If
```

Again, the condition must be true or the block of code statements will not execute. When using more than one line in the program editor for If/Then, you must end the structure with End If. The following procedure is a simple number-guessing game where the computer comes up with a number between 0 and 10 and asks the user for a guess. Three If/Then structures are used to determine what message is output to the user depending on their guess.

```
Private Sub NumberGuess()
    Dim userGuess As Integer
    Dim answer As Integer
    answer = Rnd * 10
    userGuess = Val(InputBox("Guess a number between 0 and 10.", "Number
Guess"))
    If (userGuess > answer) Then
        MsgBox ("Too high!")
        MsgBox ("The answer is " & answer)
    End If
    If (userGuess < answer) Then
        MsgBox ("Too low!")
        MsgBox ("The answer is " & answer)
    End If
    If (userGuess = answer) Then MsgBox ("You got it!")
End Sub
```

A random number generated by the Rnd function returns a random number of type single between 0 and 1. The random number is multiplied by 10 and assigned to the variable answer to make it fall between 0 and 10. Using an integer data type for the variable answer ensures that the calculated value is rounded and stored as an integer.

The If/Then structures each use one condition that compares the values stored in the userGuess and answer variables. Only one of these conditions can be true, and the message box in the If/Then structure with the true condition executes.

HINT

Previously, you saw the = operator used as an assignment operator. For example, a value is assigned to a variable. In the context of conditional expressions, the = operator is a comparison operator. Using the same character for more than one type of operation is known as over-loading an operator.

If you know you want one block of code executed when a condition is true and another block of code executed when the same condition is false, then use the Else keyword.

```
If (condition)
    'This block of code executes if the condition is true
Else
    'This block of code executes if the condition is false.
End If
```

The If/Then structures in the number guess procedure can also be written as follows, where <> is the "not equal" operator:

```
If (userGuess <> answer) Then
    MsgBox ("Wrong! The answer is "& answer)
Else
    MsgBox ("You got it!")
End If
```

This time, instead of using additional If/Then statements, the keyword Else is used to direct the program to another block of code that is executed if the condition (userGuess <> answer) evaluates to false. There is no limit on the number of conditions you can use with an If/Then code structure. The condition

```
If (userGuess <> answer) Then
```

can also be written as

```
If (userGuess < answer) Or (userGuess > answer) Then
```

Where the logical operator Or is used in the expression for the conditional. Thus, if only one conditional evaluates as true, then the expression returns true and the logic is maintained. You can use more than two conditionals if needed; however, your code will get harder to read as the number of conditionals in one line of code increases.

There are numerous possibilities for achieving the same logic when using If/Then/Else and conditionals. You can also nest the If/Then/Else code structure if you want to. The procedure below outputs a short message to the user depending on the current time and day of the week. After some variable declarations, a few familiar date functions are used to determine the current time and day of the week.

```
Private Sub myTime()
    Dim time As Date
    Dim theHour As Integer
    Dim theDayOfTheWeek As Integer
    time = Now
    theHour = Hour(time)
    theDayOfTheWeek = Weekday(time)
    If (theHour > 8) And (theHour < 17) Then
        If (theDayOfTheWeek > 0) And (theDayOfTheWeek < 6) Then
            MsgBox ("You should be at work!")
        Else
            MsgBox ("I love weekends.")
        End If
    End If
```

```
        End If
    Else
        MsgBox ("You should not be at work!")
    End If
End Sub
```

The first If/Then/Else structure is checking if the time of the day is between 8:00 A.M. and 5:00 P.M., since the variable theHour holds an integer value between 0 and 23. If the expression is true then another If/Then/Else structure will execute. This If/Then/Else structure is *nested* in the first one and is checking the value for the day of the week. If the day of the week is Monday through Friday, then a message box is used to display the string "You should be at work!". (Remember that it had to be between 8:00 A.M. and 5:00 P.M. to get to this point.) Otherwise, the nested If/Then/Else outputs the message "I love weekends." If the time of day is not between 8:00 A.M. and 5:00 P.M., then the string "You should not be at work!" is displayed in a message box.

There is no limit to the number of nested If/Then statements you can use; however, after three or four levels, keeping track of the logic can be difficult and your program may be difficult to read and debug.

TRICK

It is a good idea to indent your code with each level of logic. You will find your programs much easier to read and debug if indented properly.

Another option regarding If/Then/Else structures is the ElseIf clause. The ElseIf clause is used like the Else clause with a conditional expression. You must also include Then when using ElseIf. The following example uses a series of ElseIf clauses to display the day of the week in a message box.

```
If (theDayOfTheWeek = 0) Then
    MsgBox ("It's Sunday!")
ElseIf (theDayOfTheWeek = 1) Then
    MsgBox ("It's Monday!")
ElseIf (theDayOfTheWeek = 2) Then
    MsgBox ("It's Tuesday!")
ElseIf (theDayOfTheWeek = 3) Then
    MsgBox ("It's Wednesday!")
ElseIf (theDayOfTheWeek = 4) Then
    MsgBox ("It's Thursday!")
ElseIf (theDayOfTheWeek = 5) Then
    MsgBox ("It's Friday!")
Else
    MsgBox ("It's Saturday!")
End If
```

There is no limit to the number of ElseIf clauses that can be used; however, ElseIf cannot be used after an Else clause. You can also nest more If/Then/Else structures inside an ElseIf clause.

Select/Case

There are innumerable ways to accomplish the same task with If/Then/Else and ElseIf code structures. But keep in mind that using a large number of If/Then/Else and ElseIf statements can make it difficult to follow the logic of your program. You should consider using the Select/Case code structure in situations where you find yourself using a large number of ElseIf statements. The Select/Case code structure is used when you need to test the value of a variable multiple times and, based on the outcome of those tests, execute a single block of code. The Select/Case syntax is fairly simple and easy to understand.

```
Select Case expression
  Case condition1
    'This block of code executes if condition1 is true.
  Case condition2
    'This block of code executes if condition2 is true.
    'There is no limit on the number of cases you can use
  Case Else
    'This code executes if none of the other conditions were true.
End Select
```

A Select/Case structure must begin with Select Case and end with End Select. The expression immediately following Select Case is typically a variable of numerical or string data type. Next, a list of one or more code blocks is entered just beneath the keyword Case and a condition. The condition is a comparison to the expression in the opening line of the structure. VBA proceeds down the list until it finds a condition that evaluates as true, then executes the block of code within that case element. Any additional case elements following one that evaluates as true are ignored, even if their conditions are also true. Thus, order of the case elements is important. The last case element should use Case Else. This ensures that at least one block of code executes if all other conditions are false.

The following example uses a Select/Case structure in a VBA function designed to work with an Excel spreadsheet. The input value should be numerical and expressed as a percentage. This percentage represents a student's score and is passed into the function and stored in the variable studentScore. The variable studentScore is used as the test expression for the Select/Case structure.

```
Public Function AssignGrade(studentScore As Single) As String
  Select Case studentScore
    Case 90 To 100
      AssignGrade = "A"
    Case Is >= 80
      AssignGrade = "B"
    Case 70 To 80
      AssignGrade = "C"
    Case Is >= 60
      AssignGrade = "D"
    Case Else
      AssignGrade = "F"
  End Select
End Function
```

There are two forms for writing the conditionals in the case elements; both are shown in this example. The first case element uses Case 90 To 100. This condition is specified as a range of values with the lower value inserted first followed by the To keyword and then the upper value of the range. This condition evaluates as true if the value stored in the variable studentScore is greater or equal to 90 and less than or equal to 100.

If the value of studentScore is less than 90, VBA proceeds to the next case element which is Case Is >= 80. This is the other form for a condition using the Is keyword to specify a range with a comparison operator >= (greater than or equal to). If the value of studentScore is greater than or equal to 80, this condition is true and the block of code within this element executes (assuming the previous condition was false). Again, VBA proceeds down the list until it finds a true condition and then evaluates that case element's code block. If Case Is >= 60 in the AssignGrade() function is placed at the top of the Select/Case structure, then all students with a percentage higher than 60 would be assigned a grade of D even if they have a score of 100.

1.6 Loops and Arrays

Overview

In "Procedures and Conditions," you started building your programming foundation with the branching structures If/Then/Else and Select/Case. In this section, you will significantly expand on that foundation by learning looping code structures and arrays. Loops and arrays are fundamental to all programming languages; they expand the capabilities of a program significantly and make them easier to write. You'll begin this chapter by looking at the different looping structures available in VBA before moving on to arrays.

Specifically, this chapter will cover:

- Do Loops
- For Loops
- Arrays
- Multi-Dimensional Arrays
- Dynamic Arrays

1.6.1 Looping with VBA

Program looping is the repetition of a block of code a specified number of times. The number of times the block of code is repeated may be well defined or based on a conditional statement. All computer languages contain looping structures because these structures are excellent at solving problems that would otherwise require repetitive code. Imagine a program whose function it is to search for a specific name in a column of data with one hundred entries. A program with one hundred If/Then statements testing the value of each cell for the required name will solve the problem. The program would be technically easy to create, but cumbersome to type the repetitive code and it would look awful. Fortunately, we have looping code structures to help us.

HINT

Each execution of the block of code inside a looping structure represents one iteration of the loop.

1.6.1.1 Do Loops

Do loops will execute a given block of code repetitively based on the value of a conditional expression. All Do-Loops require the keywords Do and Loop, plus one additional keyword (While or Until) depending on the desired action. The keywords are used to build four basic representations of the Do-Loop. The first two representations use the keyword Until with a conditional statement that determines if, and how many times the code inside the loop executes. With the conditional statement at the end of the loop, the code inside the loop executes at least one time.

```
Do
    'Block of code executes at least once and continues to loop if condition is
    False'
Loop Until (condition)
```

When the conditional statement is at the beginning of the loop, the code inside the loop will not execute unless the logic of the conditional statement allows it. When using Until, the code inside the loop executes if the conditional statement is false.

```
Do Until (condition)
    'Block of code executes only if condition is false.
Loop
```

The next two representations of the Do-Loop use the keyword While with a conditional statement that determines if, and how many times the code inside the loop executes. When While is used, the code inside the loop executes when the conditional statement is true.

```
Do
    'Block of code executes at least once and continues to loop if condition is
    true.
Loop While (condition)
```

When deciding on which representation of the Do-Loop to use, ask yourself whether you need the code inside the loop to execute at least once. If you do, then put the conditional at the end. The choice of While or Until depends on the logic of the conditional expression.

```
Do While (condition)
    'Block of code executes only if condition is true.
Loop
```

Beware of creating loops that never stop repeating, otherwise known as *infinite loops*. When constructing your Do-Loop, create it with a conditional expression that will change its logical value (true to false and vice versa) at some point during the code's execution within the loop. It is easier to create an infinite loop than you might think. The following example is suppose to find the first occurrence of the string Flintstone in the first column of a worksheet, output a message to the screen, and then quit.

```
Dim I As Integer
I = 1
Do
    If (Cells(I, "A").Value = "Flintstone") Then
        MsgBox ("Yabba Dabba Do! I found a Flintstone in row "& Str(I))
    End If
    I = I + 1
Loop Until (Cells(I, "A").Value = "Flintstone")
```

TRICK

You can use the Cells property to return all or just one cell on a worksheet. Using the Cells property without any parameters returns all cells on the worksheet.

```
ActiveSheet.Cells
```

To return a specific cell, you can specify a row and column index. For example, the following line of code returns cell D1.

```
ActiveSheet.Cells(1,4)
```

The Cells property is convenient for using inside of loops when the indices for the row and column are replaced with looping variables. Alternatively, you can specify the column parameter with a string.

```
ActiveSheet.Cells(1," D")
```

The loop will always fail for two reasons. First, if the string Flintstone does not appear in the first column of the worksheet, then the loop is infinite because the conditional statement at the end of the loop (Cells(I, "A").Value = "Flintstone") will never be true. Second, even if the string Flintstone does appear in the first column of the worksheet, the output from the MsgBox() function will not

appear because the conditional statement at the end of the loop will be true before the conditional statement associated with the If/Then structure.

TRICK

If you find your program stuck in an infinite loop, use Ctrl-Alt-Break to suspend program execution.

In most cases you can construct a loop with logical expressions that will work with both While or Until, so using one or the other is simply a matter of personal preference. The following Do-Loops have the exact same function, but the first loop uses While and the second uses Until.

```
Dim I As Integer
I = 1
Do
    If (Cells(I, "A").Value = "Flintstone") Then
        MsgBox ("Yabba Dabba Do! I found a Flintstone in row "&
Str(I))
    End If
    I = I + 1
Loop While (Cells(I, "A").Value <> "")
```

If I change the conditional operator to =, then I change the logic of the conditional statement, so I must use the keyword Until to get the same result from the loop.

```
Dim I As Integer
I = 1
Do
    If (Cells(I, "A").Value = "Flintstone") Then
        MsgBox ("Yabba Dabba Do! I found a Flintstone in row "&
Str(I))
    End If
    I = I + 1
Loop Until (Cells(I, "A").Value = "")
```

Both of these loops search the first column for the string Flintstone. Once the desired string is found, a message box outputs a statement with the index of the worksheet row in which the string was found. In both examples, the Do-Loop continues until an empty cell is found. Both loops will execute at least once because the conditional expression is at the end of the loop. Neither loop will be infinite because Excel will always add empty rows to the end of a spreadsheet as more rows of data are added.

1.6.1.2 For Loops

When you know the number of iterations required from a loop, the For/Next loop is the best choice of structures. The syntax is very simple.

```
For variable = start To end Step value
    'Block of code
Next variable
```

The required keywords are For, To, and Next. To keep track of the number of iterations through the loop requires a counting variable as well as starting and ending values. The keyword Step is optional but if it's used, the value that follows it is used to denote the step size of the counting variable with each iteration through the loop. The step's value can be any positive or negative integer; the default value is +1 when Step is omitted. Table 1.8 lists a few examples of For/Next loops.

Table 1.8 – Examples of FOR /NEXT loops in VBA

Loop Example	Output from Message Box
For I = 0 To 10 MsgBox (I) Next I	11 iterations: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10
For I = 0 To 10 Step 2 MsgBox (I) Next I	6 iterations: 0, 2, 4, 6, 8, and 10
For I = 0 To 10 Step 3 MsgBox (I) Next I	4 iterations: 0, 3, 6, and 9
For I = 10 To 0 Step -5 MsgBox (I) Next I	3 iterations: 10, 5, and 0

The variable I in Table 1.8 should be declared as an integer prior to use and the ending value for the loop is usually another variable rather than a constant. In most cases, you will use the default step size of +1, so the keyword Step is omitted.

TRICK

Use the statement Exit Do or Exit For to force code execution to leave a looping structure and proceed with the first line of code after the loop. Normally, Exit Do or Exit For will be within a branching structure (If/Then or Select/Case) inside of the loop.

The following example of a VBA function mimics the FACT() function in the Excel application by calculating the factorial of an integer.

```
Public Function Factorial(myValue As Integer) As Long
    Dim I As Integer
    Dim factorialValue As Long

    factorialValue = 1
    For I = 2 To myValue
        factorialValue = factorialValue * I
    Next I
    Factorial = factorialValue
End Function
```

In the Real World

The factorial function can also be written as a recursive procedure. A *recursive* procedure is one that calls itself.

```
Public Function Factorial(N As Integer) As Integer
    If N <= 1 Then
        Factorial = 1
    Else
        Factorial = Factorial(N - 1) * N
    End If
End Function
```

Although the factorial example above is a nice illustration of recursion, it is not a practical example. Recursive procedures can be very demanding on system resources and they must contain logic that

will eventually stop the procedure from calling itself. Recursive procedures are most often and most effectively applied to tree-like data structures such as the file system on a computer.

The For/Next loop is a natural choice, because you need the looping variable to increment by one with each iteration until it reaches the value of the integer passed into the function. Each iteration through the For/Next loop multiplies the next factor by the previous result, effectively producing the factorial of the value stored in the variable myValue . For example, if myValue is 5 then the variable factorialValue will be calculated as $1*2*3*4*5$.

Finally, consider the most obvious example of looping in spreadsheet applications, which is looping through a range of cells in a worksheet. For now, I will illustrate looping through a worksheet range using a For/Next loop.

```
For I = 1 To 10
    For J = 4 To 7
        Cells(I, Chr(64 + J)).Value = I * J
    Next J
Next I
```

HINT

The looping structures discussed so far are not the best choice for looping through a range of cells—even though doing so is a simple enough task. A better looping structure for handling this task is the For/Each loop discussed later in "Basic Excel Objects."

The example above uses one For/Next loop nested inside another For/Next loop to loop through the worksheet range D1:G10. The nested (inside) loop will execute 4 iterations with each iteration of the outer loop. In the example just given, the value of J iterates from 4 through 7 for each value of I. The code loops through the range by rows, as the variable used for the row index (I) is also the counting variable for the outer loop. The Chr() function is used to convert a numerical input representing an ASCII (American Standard Code for Information Interchange) value to its corresponding keyboard character; in this case the values 68 through 71 will be converted to the uppercase letters D through G. The Chr() function in VBA works with values 0–255.

Table 1.9 - Selected ASCII conversion characters

ASCII Value	Keyboard Character
8	backspace
9	tab
10	line feed
13	carriage return
32	space
48–57	0–9
65–90	A–Z
97–122	a–z

Table 1.9 lists a few of the more common characters in the set. Alternatively, you could replace the Chr() function with the looping variable J; which, in this case, would make for easier and cleaner code; however, I wanted to introduce the Chr() function since it can be quite useful when working with the Cells and Range properties.

1.6.2 ARRAYS

Normally, arrays are not discussed until the end of introductory programming books; however, as you are already familiar with spreadsheet applications, the concept of an array should come easily. An *array* is a variable that can hold multiple values. You should use arrays when a related set of values is to be stored in a variable. Doing so relieves you from having to declare a new variable with a unique name for each value in the set. Arrays are convenient as they simplify programming code tremendously.

A spreadsheet column that contains data is basically the same thing as an array—it's a group of related values. Each cell within a spreadsheet column containing the related set of values is referenced by a row and column index. Values in an array are also referenced using indices.

I assume that you organize your spreadsheets in the normal way—by placing data inside columns rather than rows—but the argument is the same whether you equate a spreadsheet column or row to an array.

Before starting with the simplest example of an array (the one-dimensional array), consider a sub procedure that uses a worksheet column much as a programmer would use an array in an application that does not work with a spreadsheet.

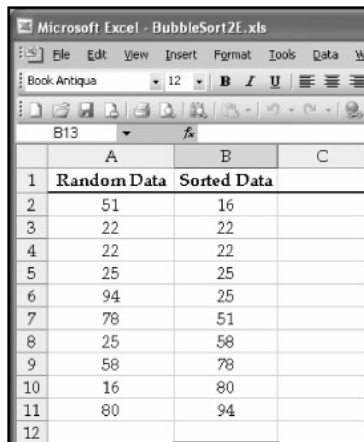
HINT

In previous sections, and throughout this section I use the Cells property of the Excel Application object in code examples. The Cells property is straightforward, with a row and column index that corresponds to a single spreadsheet cell. As you look at the examples in this chapter that the Cells property acts like a function that returns a Range object consisting of a single spreadsheet cell. I have used the Value property of the Range object extensively thus far, but the Range object has many other properties for the VBA programmer to use besides the Value property, and you will see many examples.

The BubbleSort() procedure sorts a column of integer values from lowest to highest value. Two integer variables and a Boolean variable are all you need.

```
Public Sub BubbleSort()  
'Sorts data in A2:A11 and writes sorted data to B2:B11  
    Dim tempVar As Integer  
    Dim anotherIteration As Boolean  
    Dim I As Integer  
  
    Range("A2:A11").Copy Range("B2:B11")      'Copy all data to column B  
    Range("B1").Value = "Sorted Data"  
    Do  
        anotherIteration = False  
        For I = 2 To 10  
            'Compare and swap adjacent values  
            If Cells(I, "B").Value > Cells(I + 1, "B").Value Then  
                tempVar = Cells(I, "B").Value  
                Cells(I, "B").Value = Cells(I + 1, "B").Value  
                Cells(I + 1, "B").Value = tempVar  
                anotherIteration = True  
            End If  
        Next I  
    Loop While anotherIteration  
End Sub
```

A For/Next loop nested inside a Do-Loop will iterate through a column of 10 values until the data is sorted from lowest to highest value. The nested For/Next loop effectively pushes the largest value from wherever it is located to the last position, much like a bubble rising from the depths to the surface. The For/Next loop starts at the beginning of the data list and compares two successive values. If the first value is larger than the second value, then the position of the two values are swapped with help from the variable tempVar. The next two values are then compared, where the first of these values was the second value in the previous comparison (or first if it had been swapped). Please note: the row index in the Cells property uses $I + 1$, so the looping variable in the For/Next loop works from 2 to 11 so that the procedure sorts ten values. If a swap of two values has to be made, then the Boolean variable anotherIteration is set to true to ensure the outer Do-Loop continues with at least one more iteration.



	A	B	C
1	Random Data	Sorted Data	
2	51	16	
3	22	22	
4	22	22	
5	25	25	
6	94	25	
7	78	51	
8	25	58	
9	58	78	
10	16	80	
11	80	94	
12			

Each iteration through the Do-Loop moves the next largest value in the set down the column to its correct position. Thus, it will take up to n iterations to sort the data, where n is the number of values in the set. This does not make the BubbleSort() procedure terribly efficient, but it works well for small data sets. The worksheet shown in Figure 1.24 illustrates what happens to a set of numbers after each iteration through the Do-Loop loop. Note that Figure 1.24 was created for display only; the BubbleSort() procedure sorts values from column A and copies them to column B only.

Figure 1.24 - Worksheet illustration of the BubbleSort() sub procedure.

1.6.2.1 One-Dimensional Arrays

An array is a variable used to hold a group of related values; it must be declared just as a variable is declared. An array is declared with a single name and the number of elements (values) that can be stored in the array.

```
Dim myArray(number of elements) As Type
```

You may also declare arrays using the Public or Private keywords to define the scope as you would with a regular variable declaration. If you do not specify a data type, then, like a variable, the array will be a variant type. Arrays may be declared as any available data type in VBA. All elements in arrays with numerical data types are initialized with the value 0. Elements of string arrays are initialized with an empty string. When specifying the number of elements, you must consider the lower bound of the array. The default lower bound is zero.

```
Dim myArray(10) As Integer
```

HINT

When you need multiple array declarations of the same size, use a constant to specify the size of the arrays in the declarations.

```
Const ARRAYSIZE=10
Dim myArray1(ARRAYSIZE) As Integer
Dim myArray2(ARRAYSIZE) As Integer
Dim myArray3(ARRAYSIZE) As Integer
Etc.
```

This way, if you have to edit the size of your arrays, you only need to change the value of the constant.

Thus, the integer array myArray declared above has 11 elements accessed with the indices 0 through 10. To override the default, set the lower bound of the array in the declaration

```
Dim myArray(1 To 10) As Integer
```

The array myArray now has just 10 elements because the lower bound has been explicitly set to one.

HINT

Use the statement **Option Base 1** in the general declarations section of a module to change the default lower bound of all arrays declared in the module to 1.

You can initialize a single element in the array as you would a variable, but you must include the index of the element you wish to change.

```
myArray(5) = 7
```

However, arrays are typically initialized inside a loop. To insert the spreadsheet's values of the first 10 cells of column A into an array, do the following:

```
Dim I As Integer
Dim myArray(10) As Integer
For I = 0 To 9
    myArray(I) = Cells(I + 1, "A").Value
Next I
```

Then use another loop to output the values of the array. The following loop squares the values stored in the array myArray before copying them to column B of the spreadsheet.

```
For I = 0 To 9
    Cells(I + 1, "B").Value = myArray(I)^2
Next I
```

Now let's revisit the BubbleSort() procedure, this time using an array. The sub procedure BubbleSort2() works exactly like the BubbleSort() procedure, except that the tests and swaps are performed on the values in the set after they have been loaded into an array rather than just using the worksheet column.

```
Public Sub BubbleSort2()
    Dim tempVar As Integer
    Dim anotherIteration As Boolean
    Dim I As Integer
    Dim myArray(10) As Integer
    For I = 2 To 11
        myArray(I - 2) = Cells(I, "A").Value
    Next I
    Do
        anotherIteration = False
        'Compare and swap adjacent values
        For I = 0 To 9
            If myArray(I) > myArray(I + 1) Then
                tempVar = myArray(I)
                myArray(I) = myArray(I + 1)
                myArray(I + 1) = tempVar
                anotherIteration = True
            End If
        Next I
    Loop While anotherIteration = True
```

```
Range("B1").Value = "Sorted Data"
For I = 2 To 11
    Cells(I, "B").Value = myArray(I - 1)
Next I
End Sub
```

After variable declarations, the values in column A of the worksheet are loaded into the array with a simple For/Next loop. The For/Next loop nested in the Do-Loop is just as it was in the BubbleSort() procedure, except now the Cells property has been replaced with the array named myArray. The looping variable in the For/Next loop now runs from 0 to 9 because the lower bound for the array is 0 not 1. When the first value is greater than the second, the values are swapped. Finally, the sorted values are written to column B in the worksheet.

1.6.2.2 Multi-Dimensional Arrays

If one-dimensional arrays are analogous to a single column in a spreadsheet, then twodimensional arrays are analogous to multiple columns in a spreadsheet. Three-dimensional arrays are analogous to using multiple worksheets and higher dimensions than three are a bit difficult to imagine, but nevertheless are available. You can declare multi-dimensional arrays in VBA with up to 60 dimensions. Unless you're comfortable imagining multi-dimensional spaces greater than dimension three, I suggest keeping the number of dimensions in an array to three or less.

```
Dim myArray(10, 2) As Integer
```

The above declaration creates a two-dimensional integer array with 11 rows and 3 columns (remember the lower-bound is 0). Access the individual elements of the array using the row and column indices.

```
myArray(5, 1) = Cells(6, "B").Value
```

This example assigns the value of the spreadsheet cell B6 to the sixth row and second column in the array myArray. As with one-dimensional arrays, multi-dimensional arrays are typically accessed within loops; however, you need to use nested loops in order to access both indices in a multidimensional array.

The sub procedure below transposes the values of a group of cells in a worksheet. This sub procedure takes input from the first ten rows and three columns in a worksheet and transposes the values to the first three rows and ten columns in the same worksheet. See

Figure 1.25 for depictions of the initial spreadsheet and the spreadsheet resulting from running the Transpose() sub procedure.

After variable declarations, the values in the spreadsheet are loaded into the two-dimensional array named transArray.

HINT

A three-dimensional array is declared with three values within the parentheses of its declaration (for example, Dim myArray(9, 2, 2)). You could use a threedimensional array to keep track of rows and columns from multiple worksheets, whereas a two-dimensional array would keep track of rows and columns from a single worksheet.

The figure consists of two side-by-side screenshots of the Microsoft Excel interface, both titled 'Microsoft Excel - Transpose2E.xls'. The left screenshot shows a worksheet with data in columns A, B, and C. The right screenshot shows the same worksheet after the Transpose() procedure has been executed, with the data transposed into rows 1, 2, and 3.

	A	B	C	D
1	61	21	43	
2	75	76	92	
3	98	33	50	
4	63	75	26	
5	94	77	39	
6	93	26	50	
7	65	61	59	
8	54	24	5	
9	61	81	17	
10	98	32	57	
11				

	A	B	C	D	E	F	G	H	I	J
1	61	75	98	63	94	93	65	54	61	98
2	21	76	33	75	77	26	61	24	81	32
3	43	92	50	26	39	50	59	5	17	57

Figure 1.25 - Transpose() sub procedure example

The looping variables in the nested For/Next loops are used to access the row and column indices of the array transArray. The looping variables I and J are used as the column and row indices, respectively, in both the array and worksheet. Next, the contents of the worksheet are cleared using the ClearContents method of the Range object.

To transpose the values, the looping variables I and J are now used to access the opposite index (i.e., I is used for the row index; J is used for the column index) in the Cells property; however, the array transArray uses the indices as in the previous For/Next loop. These nested For/Next loops effectively transpose the values, as shown in

Figure 1.25.

```
Public Sub Transpose()
'Transposes first 10 rows and first 3 columns of worksheet
'to first 3 rows and first 10 columns.
    Dim I As Integer
    Dim J As Integer
    Dim transArray(9, 2) As Integer
    For I = 1 To 3
        For J = 1 To 10
            transArray(J - 1, I - 1) = Cells(J, I).Value
        Next J
    Next I
    Range("A1:C10").ClearContents
    For I = 1 To 3
        For J = 1 To 10
            Cells(I, J).Value = transArray(J - 1, I - 1)
        Next J
    Next I
End Sub
```

1.6.2.3 Dynamic Arrays

The BubbleSort2() and Transpose() sub procedures use arrays with fixed lengths. The number of values in fixed length arrays cannot be changed while the program is running. This is fine as long as the required length of the array is known before running the program; however, the use of dynamic arrays allows programmers to create a more robust program. Wouldn't the BubbleSort2() procedure be more useful if it sorted data with any number of values rather than just ten values? A similar

question can be asked of the Transpose() procedure —wouldn't it be more useful if it worked with any size data set rather than just a set with 10 rows and 3 columns? If you do not want to limit the BubbleSort2() and Transpose() sub procedures to constant-sized data sets, then you must use dynamic arrays.

The size of a dynamic array can be changed (increased or decreased) as necessary while the program runs. To declare a dynamic array, use empty parentheses instead of a value for the bound(s).

```
Dim myArray() As Integer
```

After the required length of the array has been determined then the array is re-dimensioned using the ReDim keyword.

HINT

ReDim can also be used as a declarative statement with arrays, but potential conflicts may arise if there are variables of the same name within your project—even if they are of different scope. Therefore, avoid using ReDim as a declarative statement, but use it to re-size previously declared arrays.

```
ReDim myArray(size)
```

The ReDim statement will re-initialize (erase) all elements of the array. If you need to preserve the existing values then use the Preserve keyword.

```
ReDim Preserve myArray(size)
```

If the new size of the array is smaller than the original size, then the values of the elements at the end of the array are lost. Normally, an array is re-dimensioned with the Preserve keyword only when the new size is larger than the previous size of the array. When re-sizing an array with the Preserve keyword, you can only change the size of the last dimension; you cannot change the number of dimensions, and you can only change the value of the upper bound.

The BubbleSort2() and Transpose() sub procedures are now rewritten using dynamic arrays.

```
Public Sub DynamicBubble()
    Dim tempVar As Integer
    Dim anotherIteration As Boolean
    Dim I As Integer
    Dim arraySize As Integer
    Dim myArray() As Integer

    '_____
    'Get the array size.
    '_____
    Do
        arraySize = I
        I = I + 1
    Loop Until Cells(I, "A").Value = ""
    ReDim myArray(arraySize - 1)

    '_____
    'Get the values. Convert text to numbers.
    '_____
    For I = 1 To arraySize
        myArray(I - 1) = Val(Cells(I, "A").Value)
```



```

Next I
Do
    anotherIteration = False
    For I = 0 To arraySize - 2
        If myArray(I) > myArray(I + 1) Then
            tempVar = myArray(I)
            myArray(I) = myArray(I + 1)
            myArray(I + 1) = tempVar
            anotherIteration = True
        End If
    Next I
Loop While anotherIteration = True

'-----
'Write data to column B.
'-----
For I = 1 To arraySize
    Cells(I, "B").Value = myArray(I - 1)
Next I
End Sub

```

After declaring the dynamic array, you must determine the required size of the array. A Do-Loop is used to iterate through the cells in the worksheet's column A until an empty cell is found. By keeping track of the number of iterations with the variable I, the number of values in the column—and hence the required size of the array—is discovered. Then the array is redimensioned with the appropriate variable and ReDim statement.

This is not the best method for learning how many values the user has entered into column A of the worksheet, as the potential for error is high. For example, any text entered into a cell will be converted to a numerical value with the Val() function—usually zero. The procedure also limits the sort to data entered into column A of the worksheet.

The rest of the DynamicBubble() procedure is the same as the BubbleSort2() procedure except the upper limit of all looping variables are set to the same value as the size of the array.

The DynamicTranspose() sub procedure is re-written using a dynamic array that is re-dimensioned with two dimensions. One dimension is for the number of rows in the grid of values to be transposed and the other dimension is for the number of columns.

Once again, Do-Loops are used to determine the number of rows and columns holding values in the worksheet. The array transArray is then re-dimensioned to the same number of rows and columns. Don't forget the lower bound on each dimension is 0. The rest of the procedure is the same, with the exception of the upper limit on the looping variables used in the For/Next loops.

```

Public Sub DynamicTranspose()
    Dim I As Integer
    Dim J As Integer
    Dim transArray() As Integer
    Dim numRows As Integer
    Dim numColumns As Integer

    '-----
    'Get rows for dynamic array.
    '-----
    Do
        numRows = I
        I = I + 1
    
```

```

Loop Until Cells(I, "A").Value = ""

'-----
'Get columns for dynamic array.
'-----
I = 0
Do
    numColumns = I
    I = I + 1
Loop Until Cells(1, Chr(I + 64)).Value = ""
ReDim transArray(numRows - 1, numColumns - 1)

'-----
'Copy data from worksheet to array.
'-----
For I = 1 To numColumns
    For J = 1 To numRows
        transArray(J - 1, I - 1) = Val(Cells(J, Chr(I + 64)).Value)
    Next J
Next I
Range("A1:C10").ClearContents

'-----
'Copy data from array to worksheet transposed.
'-----
For I = 1 To numColumns
    For J = 1 To numRows
        Cells(I, Chr(J + 64)).Value = transArray(J - 1, I - 1)
    Next J
Next I
End Sub

```

1.6.3 Programming Formulas into Worksheet Cells

If you are going to be an Excel VBA programmer, then it is inevitable that you will have to create programs that enter formulas into worksheet cells. Thankfully, it is a pretty simple thing to do; however, you must decide on the reference style you wish to use—A1 type, or R1C1 type.

1.6.3.1 A1 Style References

The A1 style uses the column and row headings (letters and numbers, respectively) as indices to reference a particular worksheet cell (for example, A1, B5, \$C\$2, etc.). Dollar signs in front of an index denote an absolute reference; the lack of a dollar sign on an index denotes a relative reference. The A1 style reference is the preferred style of most Excel users.

Creating a formula using VBA is easy. Instead of using the Value property of the range returned by the Cells property, you use the Formula property and assign a string value. The string should be in the form of an Excel formula.

HINT

In reality, you can also assign formula strings to the Value property of a range; however, it makes your code easier to read if you use the Formula property when assigning formulas to a range.

The following example inserts a formula in cell A11 of a worksheet that calculates the sum of the values in the range A2:A10 using the Excel application's SUM() function.

```
Dim formulaString As String
formulaString = "=SUM($A$2:$A$10)"
Cells(11, "A").Formula = formulaString
```

If you want to create a set of related formulas in a column, you can use a looping structure to iterate through the cells that receive the formula. The following example uses formulas inserted into the cells of column B in a worksheet to calculate a running sum of column A.

```
Dim formulaString As String
Dim I As Integer
Cells(1, "B").Value = Cells(1, "A").Value
For I = 2 To 10
    formulaString = "=A" & Trim(Str(I)) & "+B" & Trim(Str(I - 1))
    Cells(I, "B").Formula = formulaString
Next I
```

Looping through the cells is not the most efficient method available in VBA for inserting formulas. Using loops to insert formulas can slow your program down considerably, especially if it is running on an older machine with a relatively slow processor. You would not enter individual formulas in the Excel application when it is possible to copy and paste, so why do it with your VBA code? Instead, you can use Copy() and Paste() or AutoFill() methods that run much faster.

```
Dim formulaString As String
Dim I As Integer
Cells(1, "B").Value = Cells(1, "A").Value
formulaString = "=A2+B1"
Cells(2, "B").Formula = formulaString
```

To use the Copy() and Paste() methods, first insert the formula in the original cell as before, execute the Copy() method on the range returned by the Cells property, select the desired range, and paste the formula.

```
Cells(2, "B").Copy
Range("B2:B10").Select
ActiveSheet.Paste
```

HINT

A method is yet another type of procedure that performs a specific action on a program component or *object*. The Paste() method performs its action on an Excel worksheet by pasting the contents of the clipboard onto the worksheet.

Another option is to use the AutoFill() method by specifying the destination range. The term Destination is a named argument predefined for the AutoFill() method in VBA. *Named arguments* allow the programmer to pass values to a function without having to worry about the order of the arguments, or how many commas must be included for optional arguments that are not used. Use the named argument operator (:=) to assign the value to the name.

```
Cells(2, "B").AutoFill Destination:=Range("B2:B10")
```

Or, if you prefer, you can still pass the arguments in a list.

```
Cells(2, "B").AutoFill Range("B2:B10")
```

The second line of code using the AutoFill() method works because Destination is the first argument/parameter that must be passed to the method. (As it turns out, the Destination argument is

the only required parameter of the AutoFill() method.) Using the named argument with the named argument operator makes the code more readable; therefore, the first example with the AutoFill() method is probably better. You can use named arguments with any procedure in VBA.

1.6.4 R1C1-Style References

The R1C1 style uses the letters R for row and C for column followed by numbers to reference spreadsheet cells. For example, R[-1]C[2] is a relative reference to the cell one row lower and two columns higher than the cell that contains this reference in a formula. To denote an absolute reference, leave off the brackets (for example, R-1C2). The R1C1 reference style can be turned on in the Excel application by clicking Tools, Options, General, and then clicking R1C1 reference style as shown in Figure 1.26.

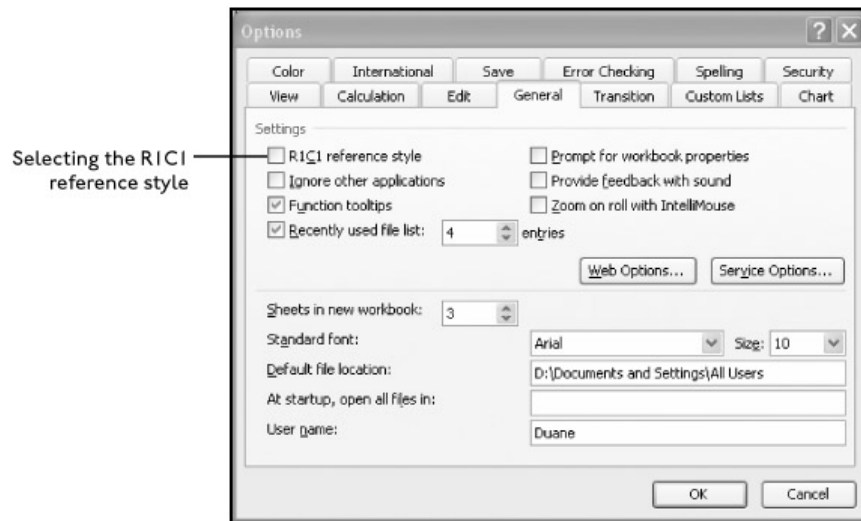


Figure 1.26 - Selecting the R1C1 reference selection in the Excel application.

You can use the R1C1 reference style in your VBA code any time. It can be a preferable style to use when dealing with references to columns, as the indices use a numerical value. The value of the string variable formulaString in the previous example can be assigned as shown here:

```
formulaString = "=R[0]C[-1]+ R[-1]C[0]"
Cells(2, "B").FormulaR1C1 = formulaString
```

Although the Formula property of the Range object returned by the Cells property would work just as well, I have used the FormulaR1C1 property for consistency.

HINT

Whether you use the A1 style or R1C1 reference style in your VBA code is of no consequence to the user. The user will see whichever style they have set their Excel application to use.

1.7 Basic Excel Objects

Overview

The preceding chapters concentrated on fundamental programming constructs common to all languages. Now it is time to introduce some VBA-and Excel-specific programming concepts and capabilities. You will be using programming tools referred to as *objects*, specifically some of the objects available in VBA and Excel.

In this section you will learn about:

- Objects
- VBA Collection Objects
- Workbook and Window Objects
- The Worksheet Object
- The Range Object
- With/End With and For/Each

1.7.1 VBA and Object-Oriented Programming

If VBA is your first programming language, then chances are you have not heard of object-oriented programming. Don't worry if you haven't heard of it; VBA does not qualify as an object-oriented language. There are some technicalities that disqualify VBA from calling itself "object-oriented," but VBA still shares many of the same concepts as genuine object-oriented languages. Mainly, object-oriented languages and VBA commonly share the existence of objects and some of the tools used to manipulate these objects. These tools include properties, events, and methods. (Other languages may call these tools something different, but they are really the same thing.) You have already seen several VBA objects in action. Objects must be discussed in VBA at a relatively early stage. Objects show up early, often, and everywhere in your VBA code. This is a good thing, because your programs can't really do much without them.

1.7.2 VBA Collection Objects

Collection objects in VBA are fairly straightforward—they are exactly what the name implies: a group or collection of the same object types. Referring to the bicycle example again, consider a collection of bicycles. The bicycle objects in your bicycle collection can be different sizes, colors, and types, but they are all bicycles. Collection objects allow you to work with objects as a group rather than just working with a single object. In VBA, collection objects are typically denoted with the plural form of the object types that can belong to a collection (not all can). For example, any Workbook object belongs to a Workbooks collection object. The Workbooks collection object contains all open Workbook objects. The Excel window shown in Figure 1.27 contains three open Workbook objects (Book1, Book2, and Book3).

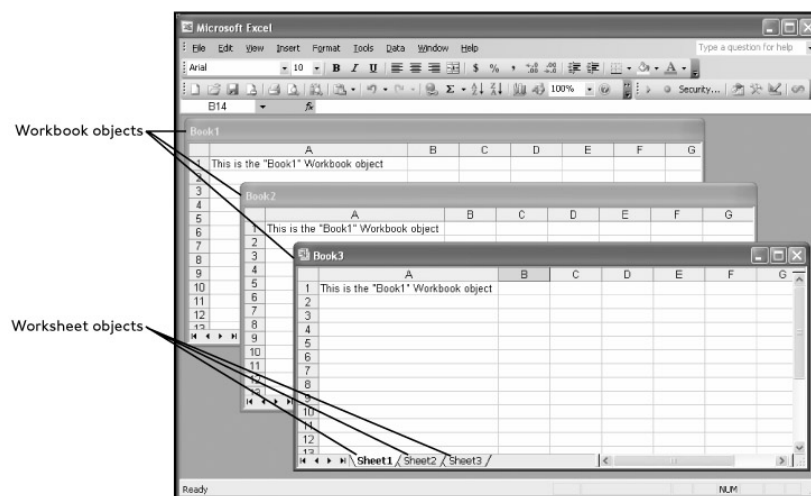


Figure 1.27 – Excel Workbook Objects

To select a Workbook object from the Workbooks collection object, the code would look like this:

```
Workbooks(2).Activate
```

This line of code uses the Workbooks property of the Application object (more on this later) to return a single Workbook object from the Workbooks collection object and then uses the Activate() method of the Workbook object to select the desired object.

HINT

The required syntax when addressing objects in VBA is object.property or object.method. You may also specify multiple properties in order to reach the desired property or method. For example, `Application.ActiveSheet.Range("A1").Font.Bold = True` is of the form object.property.property.property.property because ActiveSheet, Range("A1"), and Font all represent properties that return objects. Bold is a Boolean property of the Font object and its value is set to true. As you may have guessed, this line of code turns on bold formatting in cell A1 of the current worksheet.

So, from the collection of Workbook objects shown in Figure 1.27, which Workbook object does the previously mentioned line of code return? If you answered Book2, you'd be wrong, although that is the intuitive answer. The number in parentheses refers to a relative index number for each Workbook object as it was created. (In this case, Book1 was created first, Book2 second, and Book3 third.) The confusing part is that an index value of 1 is reserved for the currently selected Workbook object, regardless of when that Workbook object was created. So to select Book2 you would actually have to use an index value of 3 in the above line of code. An index value of 2 would return Book1 and an index value of 1 or 4 would return Book3.

There will always be two choices of an index for the currently selected Workbook object, the value 1 because it is reserved for the currently selected object, and the value corresponding to its sequence in being created. The behavior of the Workbooks collection object can be confusing, but with practice, patience, and above all, testing, I'm sure you can figure it out.

To avoid confusion, you can select a workbook unambiguously—if you know the name of the desired Workbook object—using the following line of code.

```
Workbooks("Book2").Activate
```

Here you simply include the name of the object as a string in place of the index number. Obviously, this is much less confusing and makes your code easier to read, so I recommend doing it this way whenever possible.

TRICK

When you need to step through several objects in a collection, use a loop and a looping variable to represent the index of the object to be returned.

```
For I=1 To 3  
    If Workbooks(I).Saved Then Workbooks(I).Close  
Next I
```

Other examples of collection objects include Worksheets, Windows, and Charts. For example, each of the Workbook objects in Figure 1.27 contains three Worksheet objects that belong to separate Worksheets collection objects. There are three Worksheets collection objects in this example because they are lower in the object hierarchy than the Workbook object.

1.7.3 EXCEL OBJECTS

1.7.4 The Worksheet Object

The Worksheet object falls just under the Workbook object in Excel's object hierarchy. To investigate some of the events of the Worksheet object, the following code has been added to the SelectionChange() event procedure of Sheet1 in the Center.xls workbook.

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Dim msgOutput As String
    msgOutput = "The name of this worksheet is "& Worksheets(1).Name
    MsgBox (msgOutput)
    Worksheets(2).Select
End Sub
```

The SelectionChange() event procedure is triggered whenever the user changes the current selection in the worksheet. The Target argument passed to the SelectionChange() event procedure is a range that represents the cells selected by the user. I will discuss the Range object shortly; for right now, ignore it because the current example does not use the passed argument.

The code in the SelectionChange() event procedure is straightforward. First, a string variable is created and assigned a value ("The name of this worksheet is") that is then concatenated with the name of the worksheet obtained from the Name property of the Worksheet object. The full object path is not used to return the name of the worksheet, as this code will only be executed when the user changes the selection in the first worksheet of the Worksheets collection object (Sheet1). Therefore, the object path travels through the current Workbook object.

This is why index numbers can be used with the Worksheets property of the Workbook object without having to worry about returning the wrong sheet. After displaying the concatenated string in a message box, the Select() method of the Worksheet object is used to select the second worksheet in the Worksheets collection object. (This will generate an error if only one worksheet exists in the collection.)

Next, code is added to the Worksheet_Activate() event procedure of Sheet2. The Worksheet_Activate() event procedure is triggered when a worksheet is first selected by the user or, in this case, by selecting the worksheet using program code (Worksheets(2).Select). The code is essentially the same as the previous example.

```
Private Sub Worksheet_Activate()
    Dim msgOutput As String
    msgOutput = "This worksheet is "& Worksheets(2).Name
    MsgBox (msgOutput)
End Sub
```

TRICK

The Worksheet_Activate() event procedure is not triggered when a workbook is first opened, so it is not a good place for initialization routines intended to run as soon as a workbook is opened. These procedures should be placed in the Workbook_Open() event procedure.

HINT

You may have noticed in the object browser an object called Sheets. The Sheets collection object is nearly identical to the Worksheets collection object and the two objects can often be used interchangeably (as is the case in the previous two examples). The difference between

these two objects is that the Sheets collection object will also contain any chart sheets open in the active work-book. So, if you expect chart sheets to be open in the workbook of interest, you should access worksheets using the Sheets collection object; otherwise, either collection object will suffice.

1.7.5 The Range Object

The Range object represents a group of one or more contiguous cells in an Excel worksheet. The Range object is one level beneath the Worksheet object in Excel's object hierarchy, and it is extremely useful, as it allows us to manipulate the properties of an individual cell or collection of cells in a worksheet. You will probably find yourself using the Range object in every program you write using VBA for the Excel application.

Consider the following code examples that use properties of the Range object.

```
Range("A1").Value=" Column A"  
Range("A1:G1").Columns.AutoFit  
Range("A1:C1", "E1:F1").Font.Bold = True
```

HINT

The Range object is one example of a VBA collection object that does not use the plural form of an existing object for its name. The Range object is a collection object in the sense that it represents a collection of cells in a worksheet, even if the collection represents only one cell.

First, note that a long object path is omitted from the examples above; thus, these lines of code will operate on the currently selected worksheet. The first line inserts the text Column A into cell A1 by setting its Value property. The Range property was used to return a Range object representing a single cell A1) in this example. You have already seen several examples of the Value property in this book. Although the Value property exists for several objects, it is the Range object for which it is most commonly used. The second line of code above uses the AutoFit() method of the Range object to adjust the width of columns A through G such that the contents of row 1 will just fit into their corresponding cells without overlapping into adjacent columns. This is equivalent to the user selecting Format, Column, AutoFit Selection from the Excel application menu.

Entries in other rows that are longer than the entries in row 1 will still run into the next column. To automatically adjust the width of these columns such that the contents of every cell in the columns fit within cell boundaries, use the range A:G instead of A1:G1. The third and last example demonstrates setting the Bold property of the Font object to true for two distinct ranges in the active worksheet. The two ranges are A1:C1 and E1:F1. You are allowed to return a maximum of two ranges, so adding a third range to the arguments in the parentheses would generate a run-time error.

The examples above demonstrate just a couple of formatting methods and properties belonging to the Range object (AutoFit(), Columns, and Font). If you are a regular user of Excel, then you have probably surmised that there are numerous other properties and methods related to formatting spreadsheet cells. You can either search the Object Browser or the online help for more examples on how to use formatting options of interest; however, when you know what formatting options you want to include in your VBA program, record a macro. It is a quick and easy way to generate the code you need without having to search the documentation for descriptions of the desired objects, properties and methods. After you have recorded the macro in a separate module, you can clean up the recorded code and then cut and paste into your program as needed.

You may have noticed that the range arguments used in the examples above (A1, A1:G1, etc.) are of the same form used with cell references in the Excel application. The identical syntax is highly convenient because of its familiarity.

Finally it is time to take a closer look at the Cells property, specifically the Cells property of the Application, Range, and Worksheet objects.

1.7.5.1 Using the Cells Property

The Cells property returns a Range object containing all (no indices used) or one (row and column indices are specified) of the cells in the active worksheet. When returning all of the cells in a worksheet, you should only use the Cells property with the Application and Worksheet objects, as it would be redundant, and thus confusing, to use it with the Range object. For example,

```
Range("A1:A10").Cells
```

returns cells A1 through A10, thus making the use of the Cells property unnecessary.

HINT

The Cells property will fail when using it with the Application object unless the active document is a worksheet.

To return a single cell from a Worksheet object you must specify an index. The index can be a single value beginning with the left uppermost cell in the worksheet (for example, Cells(5) returns cell E1) or the index can contain a reference to the row and column index (recommended) as shown below.

```
Cells(1, 4).Value=5  
Cells(1," D").Value =5
```

This is the familiar notation used throughout this book. Both lines of code will enter the value 5 into cell D1 of the active worksheet. You can either use numerical or string values for the column reference. You should note that the column reference comes second in both examples and is separated from the row reference by a comma. I recommend using the second example above, as there is no ambiguity in the cell reference—though on occasion it's convenient to use a numerical reference for the column index.

Now consider some examples using the Cells property of the Range object.

```
Range("C5:E7").Cells(2, 2).Value = 50  
Range("C5:E7").Cells(2, "A").Value = 50
```

This code may confuse you because they appear to be trying to return two different ranges within the same line of code; however, that is not the case, but you can use these examples to more carefully illustrate how the Cells property works.

Before reading on, guess in what worksheet cell each of these lines places the value 50. If you guessed cells B2 and A2, respectively, you're wrong. Instead, the value 50 is entered in cells D6 and A6, respectively, when using the above lines of code. Why? It's because the Cells property uses references relative to the selected range. Without the reference to the Range object in each statement (Range("C5:E7")), the current range is the entire worksheet, thus Cells(2,2) returns the range B2; however, when the selected range is C5:E7, Cells(2,2) will return the second row from this range (row 6) and the second column (column D). Using a string in the Cells property to index the column forces the selection of that column regardless of the range selected. The row index is still relative; therefore, the second example above returns the range A6.

1.7.6 Working with Objects

You have now seen numerous examples of objects and how to set their properties and invoke their methods and events, but there are a couple more tools that can be of tremendous use when working with objects: the With/End With code structure that, although never required, works well to simplify code; and the object data type, which allows you to reference existing objects or even create new objects. The object data type is not as easy to use as the numerical and string data types you're now familiar with, but it is an essential tool for the creation of useful and powerful VBA programs.

1.7.6.1 The With/End with Structure

Although the With/End With structure discussed in this section is not required under any circumstances, its use is often recommended because it makes your programs more readable. Also you will often see the With/End With structure in recorded macros. Consider the following code:

```
Range("A1:D1").Select
With Selection.Font
    .Bold = True
    .Name = "Arial"
    .Size = 18
End With
With Selection
    .HorizontalAlignment = xlCenter
    .VerticalAlignment = xlCenter
End With
```

When executed, this code selects the range A1:D1 of the active worksheet using the Select() method of the Range object. The Select() method applies to several objects including the Worksheet and Chart objects. You will notice that using the Select() method with the Range object will cause the selected range to be highlighted in the worksheet, just as if the user used the mouse to make the selection.

Immediately after invoking the Select() method, the With/End With structure appears. The With statement requires an object qualifier to immediately follow. In this case, the Selection property of the Window object is used to return a Range object from which the Font property returns a Font object associated with the selected range. The statement could have just as easily been written without the Select() method and Selection property and entered using the Range property to return the desired Range object (for example, With Range("A1:D1").Font).

Once inside the structure, any property of the object can be set without having to qualify the object in each line of code. Subordinate objects and their properties can also be accessed. Each line within the structure must begin with the dot operator followed by the property or object name, then the method or assignment.

After all desired properties and/or methods have been invoked for the given object, the structure closes with End With. You will note that a second With/End With structure is used to set the horizontal and vertical alignment of the selected range. This is because I recorded this code and cleaned it up by deleting lines of code created by the macro recorder for default assignments. The example can be compressed further as shown below:

```
With Range("A1:D1")
    .HorizontalAlignment = xlCenter
    .VerticalAlignment = xlCenter
    .Font.Bold = True
End With
```

```
.Font.Name = "Arial"  
.Font.Size = 18  
End With
```

The With/End With structure is straightforward and particularly useful when a large number of properties or methods of one object are to be addressed sequentially in a program.

1.7.6.2 For/Each and Looping Through a Range

As stated at the beginning of this chapter, objects are often built from other objects. For example a Workbook object usually contains several Worksheet objects, which in turn contains multiple Range objects. It may be necessary, on occasion, to select individual objects contained within other objects. For example, you may want to access each individual Worksheet object in a Worksheets collection object in order to set certain properties. If you are thinking loops then you are right on track, but you're not going to use any of the looping structures previously discussed. Instead, you'll use a looping structure specifically designed to iterate through collections. The loop is the For/Each loop, and its use is illustrated in the example that follows:

```
Dim myRange As Excel.Range  
Dim myCell As Excel.Range  
  
Randomize  
Set myRange = Range("A1:B15")  
For Each myCell In myRange  
    myCell.Interior.ColorIndex = Int(Rnd * 56) + 1  
Next
```

In this example, the background of a group of cells is changed to all different colors. To accomplish this, each cell is accessed individually as a Range object before setting the ColorIndex property of the Interior object. The For/Each loop is used for this purpose.

Two object references are required with the For/Each loop; one for the individual objects, and the other for the collection of objects. In this example, the object variable myRange represents a collection of cells while the object variable myCell represents each individual cell within myRange.

The reference to the object variable myRange must be set (cells A1 through B15 in this example) before it can be used in a For Each loop.

The loop begins with the keywords For Each, followed by the variable that is to represent the individual elements in the collection—myCell in this example. The keyword In is followed by the name of the collection—myRange in this example. Note, that it is not necessary to set the object reference to the variable myCell, as VBA handles this automatically in the For Each loop.

Inside the loop, properties and methods of the individual elements can be addressed. In this case, the ColorIndex property of the Interior object is changed using a randomly generated number between 1 and 56 (there are 56 colors in Excel's color palette). Once each statement within the loop is executed, the Next keyword is used to continue the loop.

VBA iterates through the cells in the collection first by row and then by column. Therefore, in this example, the order follows A1, B1, A2, B2, A3, B3, and so on.

When all elements of the collection have been accessed and each statement executed, program execution resumes at the end of the loop as normal. The above code was added to a standard module in a sub procedure named CellColors() and executed. Figure 1.28 shows the result.

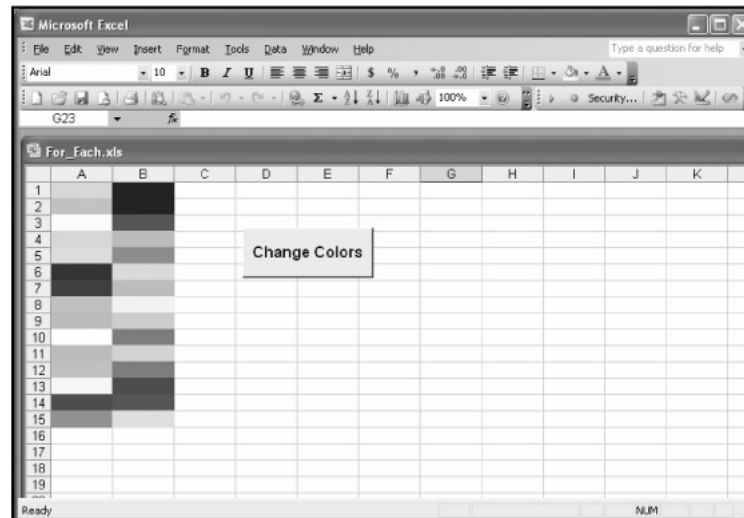


Figure 1.28 - The result of executing the CellColors() sub procedure.

1.8 Basic File I/O & Debugging

The ability to read and write data to a computer's disk drives is fundamental to most programming languages. This chapter examines some of the different tools available in VBA and Excel that allow a programmer to write code for viewing a computer's file structure, and to read and write text files. Additional tools required for error handling and debugging your VBA programs are also discussed.

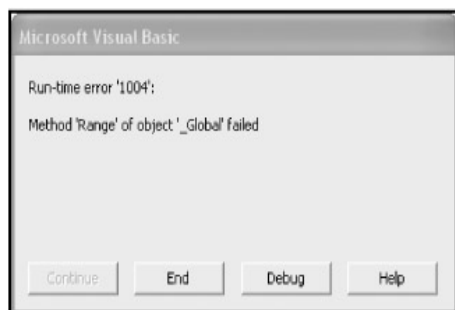
Specifically, this section will cover:

- Debugging
- File dialogs
- Creating text files

1.8.1 Debugging

By now, you have certainly encountered numerous errors in your programs and probably struggled to correct some of these errors. Finding bugs in a program can be frustrating. Fortunately, VBA has several tools to help debug a program.

1.8.1.1 Break Mode



When your program generates a runtime error, a dialog box similar to the one shown in Figure 1.29 is displayed.

Selecting the Debug option will load the VBA IDE and display the program in Break Mode. While in Break Mode, program execution is paused and can be stepped through one line at a time to closely examine factors such as order of code execution and the current values stored within variables. The line of code that generated the error will be highlighted as shown in Figure 1.30.

Figure 1.29 – The runtime error dialog box

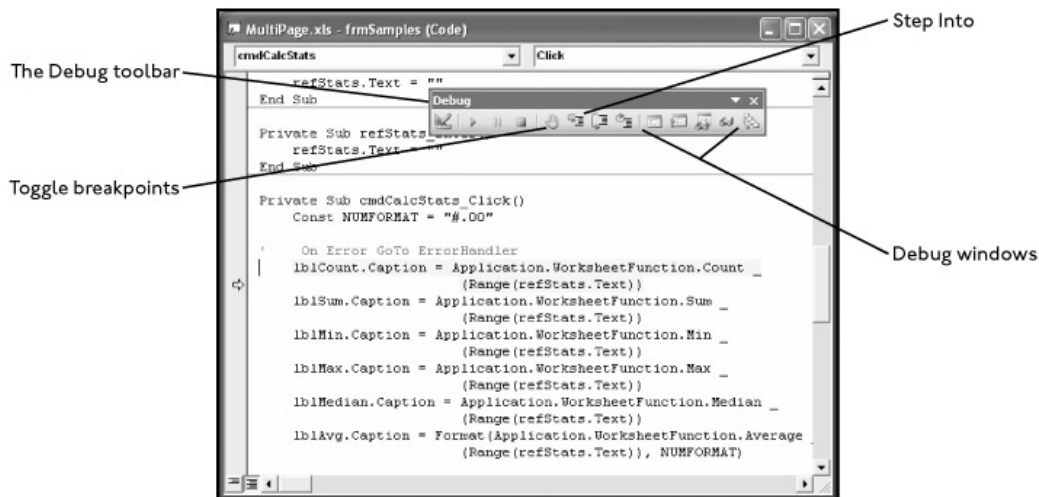


Figure 1.30 - The VBA IDE in Break Mode.

To intentionally enter Break Mode, insert a breakpoint at the desired location in the program using the Debug menu item or Debug toolbar (select from the View menu) in the VBA IDE (refer to Figure 1.30). You can also toggle a breakpoint by clicking the left margin of the code window next to the line of code at which you want program execution to pause, or by pressing F9.

Insert break points at locations in code where bugs are suspected or known to exist and then run the program. Break Mode is entered when program execution proceeds to a line of code containing a breakpoint. At this time, you have the option of resetting the program, stepping through the program one line at a time, or continuing normal operation of the program. While in Break Mode, the value currently stored in a variable can be checked by holding the mouse cursor over the name of that variable. Logic errors are often caused by code that assigns the wrong value to a variable. Break Mode can help locate the code that creates these errors.

Another useful debugging method is stepping through code while in Break Mode. Use Step Into on the Debug toolbar, or press F8, to execute one line of code at a time starting from the location of the break. The order of program execution can be verified, and values stored within variables checked as code execution proceeds one line at a time.

1.8.1.2 The Immediate Window

Stepping through code one line at a time can be tedious if the error is not found quickly. The Immediate window allows you to test program variables and procedures under normal program execution. The Immediate window is displayed by selecting it from the View menu, the Debug toolbar (refer to Figure 1.30), or by pressing Ctrl + G in the IDE.

The Immediate window is often used to hold the value of a variable or variables written to it with debugging statements located at suspected trouble spots in the program. Debugging statements use the Assert() and Print() methods of the Debug object. The Assert() method can be used to break program execution based on a Boolean expression. The Print() method is used to write values to the Immediate window.

HINT

Debugging statements are not compiled and stored in the executable program file, so there is no harm in leaving them in your code.

1.8.1.3 The Watch Window

Besides the Immediate window, another useful tool for debugging VBA programs is the Watch window. The Watch window makes it possible to track the value of a variable or expression (property, function call, and so on) from anywhere in a program. Add a watch to an expression from the Debug menu or right click the expression and choose Add Watch from the shortcut menu. The resulting dialog box is shown in Figure 1.31.

Choose either a specific procedure containing the expression you want to watch, or choose all procedures. Next, choose a specific module containing the expression you want to watch, or select all modules. Finally, select the type of Watch (Watch Expression, Break When Value Is True, or Break When Value Changes). The watch type selected will be displayed in the Watch window only when the program enters Break Mode. Therefore, if the Watch type Watch Expression is selected, a breakpoint will have to be inserted in the procedure(s) containing the expression before running the program. The other two watch types automatically pause the program at the specified location. A Watch window showing the value of an expression while the program is in Break Mode is shown in Figure 1.32.

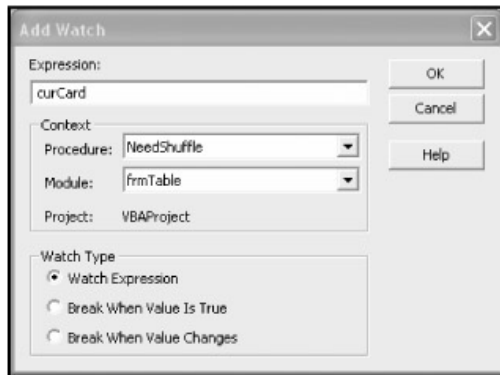


Figure 1.31 - The Add Watch dialog box.

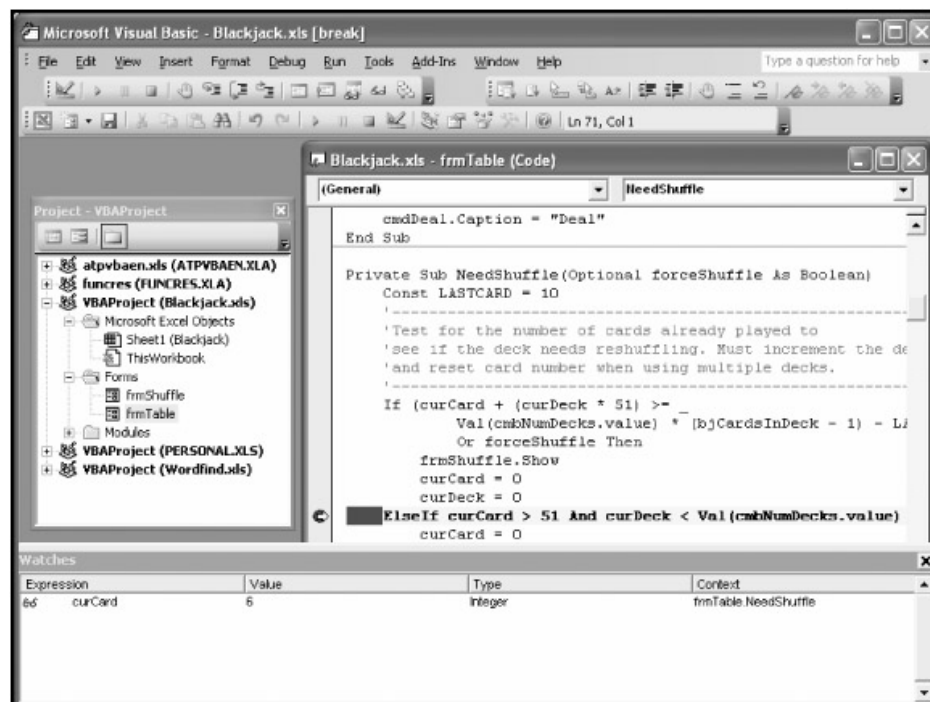


Figure 1.32 - : The Watch window.

1.8.2 File Input and Output (I/O)

VBA includes several objects, methods, and functions that can be used for file I/O. You have probably surmised that one possibility for file I/O involves the Workbook object and its methods for saving and opening files. However, there are other tools available in VBA, some of which will be discussed in this chapter.

When a VBA application requires file I/O, it often involves a relatively small amount of data stored in program variables and not in a worksheet or document. With Excel, the programmer has the choice of copying the data to a worksheet so the user can save the data in the usual way (File/Save menu item) or saving the content of the variables directly to a file. It is often more convenient to simply write the data directly to a file on the hard drive so the user does not have to be concerned with the task. In fact, it may be undesirable to give the user access to the data, as he or she might alter it before saving. In this case, reading and writing simple text files within the program code offers an attractive solution.

In the Real World

There are many types of files stored on a computer's hard drive including operating system files (for example, Windows or Macintosh files used to handle specific tasks performed by the OS), image files, and Excel files. Most of these files are created by specific applications and are therefore proprietary. Proprietary files should only be accessed by the applications from which they were created. In Windows, proprietary files have unique file extensions such as .doc, .xls, and .xcf, to name just a few. The file extensions are used by the operating system to identify the application that created the file.

A proprietary file (such as one created by Excel) contains not only the textual and numerical information entered by the user, but also content that the application uses to specify formatting options selected by the user (bold, font size and type, and so on) as well as any nontextual information entered by the user (for example, images, and charts). The methods used to write all this additional information to the file are specific to the application, and therefore can only be opened by the application that originally created the file.

1.8.2.1 Using VBA File I/O Methods

The file I/O methods discussed in this chapter are not associated with any VBA or Excel objects, although such methods do exist. The Excel object library contains several file I/O methods found with the Application and Workbook objects. Also, VBA includes several I/O objects, such as the FileSystem and FileSystemObject objects, the TextStream object, and the Drive object (among other associated objects and methods). These objects are conceptually somewhat more difficult to use than what will be discussed in this section. In this section we are going to cover the sequential access type.

Table 1.10 – File Access Modes with VBA

Access Type	Writing Data	Reading Data
Sequential	Print#, Write#	Input#, Input
Random	Put	Get

1.8.2.2 File I/O using Workbook and Worksheet Objects

The Workbook and Worksheet objects contain methods for opening, closing, and saving workbooks in the same manner a user might perform these operations from the Excel application. You can open and save workbook files using a variety of formats with VBA code. The same file formats may also be used to save individual worksheets within an existing workbook.

Opening and Saving Workbooks

You use the `Open()` method of the `Workbooks` collection object to open Excel-compatible files. The `Open()` method accepts numerous arguments, but the only required argument is the `Filename`. The syntax for the `Open()` method of the `Workbooks` collection object, including all arguments, follows:

```
Workbooks.Open(Filename, UpdateLinks, ReadOnly, Format, Password,
WriteResPassword, IgnoreReadOnlyRecommended, Origin, Delimiter, Editable,
Notify, Converter, AddToMru, Local, CorruptLoad)
```

You will never use most of these arguments, but those with unfamiliar names can be found in the online help. The following statement opens a workbook named `MyWorkbook.xls` located in the same directory as the active workbook. Note that the active workbook must be previously saved or the `Path` property of the `Workbook` object will not return a valid file path. Alternatively, you may use a string to specify the full path.

```
Dim filePath As String
filePath = ActiveWorkbook.Path
Workbooks.Open Filename:=filePath & "\MyWorkbook.xls"
```

To save a workbook from a VBA program, use either the `Save()` method of the `Workbooks` collection object or the `SaveAs()` method of the `Workbook` object. The `Save()` method does not accept arguments and will save a new workbook to the default directory (the directory last used or the directory specified in the `General` tab of Excel's `Options` dialog if a workbook has not been previously saved).

```
Workbook("MyWorkbook.xls").Save
```

The following line of code saves the active workbook to the default directory as an Excel 2003 file (`xlWorkbookNormal`).

```
ActiveWorkbook.SaveAs Filename:= "MyWorkbook.xls", FileFormat:=xlWorkbookNormal
```

TRICK

Text files only contain characters from the ANSI character set. The ANSI character set is comprised of 256 characters that represent the characters from your keyboard (alphabetical, numerical, punctuation, and so on).

```
ActiveSheet.SaveAs Filename:=" MyData.csv", FileFormat:=xlCSV
```

Figure 1.33 shows an Excel worksheet with random numerical data that has been saved as a comma-delimited text file. Figure 1.33 shows the resultant file opened in WordPad.

1.8.2.3 Using VBA File I/O Methods

In addition to the `Open()`, `Save()`, and `SaveAs()` methods of the `Workbooks`, `Workbook`, and `Worksheet` objects, VBA and its associated object libraries include several I/O objects such as the `Dialogs`, `FileDialog`, `FileSystem`, and `FileSystemObject` objects, and other subordinate objects. Some of these objects are conceptually more difficult to use and therefore will not be discussed in

this chapter; however, I will show you how to use one object from the Office library and VBA's Open statement for adding file I/O to your programs.

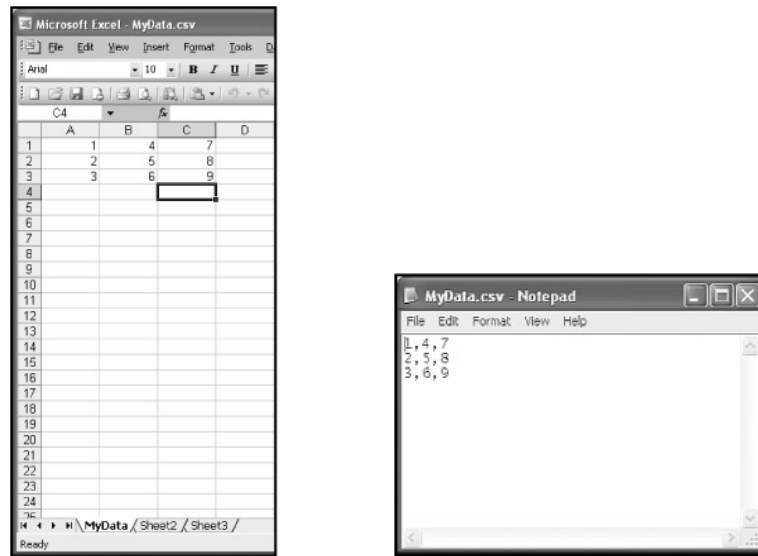


Figure 1.33 - An Excel worksheet after saving as a text file (.csv extension).

The FileDialog Object

Included in the Office library of objects is the FileDialog object. The FileDialog object is essentially the standard dialog used in Office applications for opening and saving files. The dialog boxes from the FileDialog object allow users to specify the files and folders that a program should use and will return the paths of the selected files or folders. You can also use the FileDialog object to execute the associated action of the specified dialog box.

TRICK

A reference must be set to the Microsoft Office object library before you can use the FileDialog object. From the VBA IDE, select Tools, References, and be sure the Check Box labeled Microsoft Office 11.0 Object Library is selected.

The FileDialog object contains two methods called Show() and Execute(). You use the Show() method to show one of four possible dialog boxes (see Table 1.11) depending on the constant passed to the FileDialog property of the Application object. The following statement shows the Open dialog.

Table 1.11- Dialog Types Used With The Filedialog Object

Dialog Type	VBA Constant (FileDialogType)
Open	msoFileDialogOpen
Save	msoFileDialogSaveAs
File Picker	msoFileDialogFilePicker
Folder Picker	msoFileDialogFolderPicker

The Execute() method allows the user to carry out the specified action of the dialog box for files that are compatible with the Excel application (for example, files of type .xls, .xlt, .csv, and so on). For example, the Open dialog box allows the user to select one or more files to open when the Execute() method of the FileDialog object is invoked. When the following statement follows the Show() method for the Open dialog, the item(s) selected by the user are opened in Excel.

```
Application.FileDialog(msoFileDialogOpen).Execute
```

TRICK

Be careful to set the properties of the FileDialog object appropriately for the desired action. For example, you cannot set the FilterIndex property of the FileDialog object when showing the Folder Picker dialog box because this dialog box shows only folders and does not allow file extension filters.

The FileDialogFilters and FileDialogSelectedItems Collection Objects

The FileDialog object has two subordinate collection objects—the FileDialogFilters and the FileDialogSelectedItems collection objects. The FileDialogFilters collection object contains a collection of FileDialogFilter objects that represent the file extensions used to filter what files are displayed in the dialog box (used with the Open and Save As dialog boxes). Use the Filters property of the FileDialog object to return the FileDialogFilters collection and the Item property of the FileDialogFilters collection object to return a FileDialogFilter object. The Description and Extensions properties of the FileDialogFilter object return the description (for example, All Files) and the file extension used to filter the displayed files (for example, *.*).

I wrote the CheckFileFilters() sub procedure to generate a list of all possible file filters and their descriptions, then output the lists via message boxes. The procedure simply loops through each FileDialogFilter object in the FileDialogFilters collection and concatenates their Description and Extensions properties to separate string variables. Add the following procedure to any code module then run the program to generate message boxes similar to those shown in Figure 1.34.

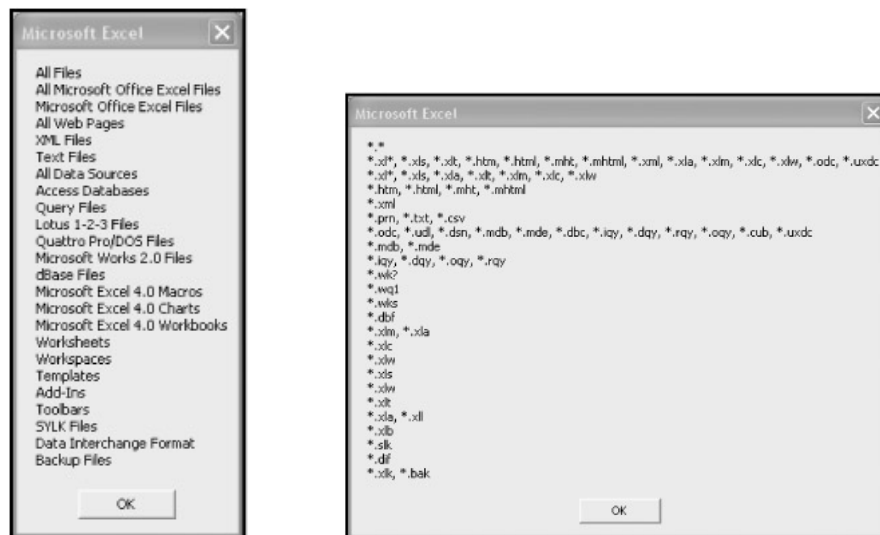


Figure 1.34 - File filter descriptions for Excel and File file extensions for Excel.

```
Public Sub CheckFileFilters()
    Dim fileFilters As FileDialogFilters
    Dim fileFilter As FileDialogFilter
    Dim I As Integer
    Dim descrs As String
    Dim xtns As String

    Set fileFilters = Application.FileDialog(msoFileDialogOpen).Filters
    '_____
    'Loop through collection and build strings of
    'all extensions and descriptions.
    '_____
```

```

    For I = 1 To fileFilters.Count
        Set fileFilter = fileFilters.Item(I)
        descrs = descrs & fileFilter.Description & vbCrLf      'Add carriage
return/line feed to strings.
        xtns = xtns & fileFilter.Extensions & vbCrLf
    Next I
    MsgBox descrs
    MsgBox xtns
End Sub

```

The `FileDialog.SelectedItems` collection object contains the paths (as strings) to the files or folders selected by the user. Use the `SelectedItems` property of the `FileDialog` object to return the `FileDialog.SelectedItems` collection. The `GetSelectedItem()` sub procedure first shows the Open dialog then loops through all items selected by the user in order to build a string containing their file paths. The file paths are then output in a message box. Note that the `Item` property of the `FileDialog.SelectedItems` object returns a string.

```

Public Sub GetSelectedItem()
    Dim selItems As FileDialog.SelectedItems
    Dim I As Integer
    Dim paths As String
    '_____
    'Build a list of file paths to all files selected by user from Open
dialog.
    '_____
    Application.FileDialog(msoFileDialogOpen).Show
    Set selItems = Application.FileDialog(msoFileDialogOpen).SelectedItems
    For I = 1 To selItems.Count
        paths = paths & selItems.Item(I) & vbCrLf
    Next I
    MsgBox paths
End Sub

```

You can use the `Add()` method of the `FileDialogFilters` collection object to create your own list of filters. The `LoadImage()` sub procedure shows the File Picker dialog box after clearing the `FileDialogFilters` collection and adding two new filters (*.*, and *.bmp). The `Add()` method requires a description and extension. An optional `Position` argument indicates the position of the added filter in the list.

The `Show()` method is called to display of the Open dialog after its properties are set. The `Show()` method of the `FileDialog` object returns -1 if the user presses the action button (Open in this example) and 0 if the action is cancelled. The `FilterIndex` property sets which filter is selected when the dialog is shown—essentially creating a default file filter. With the `AllowMultiSelect` property of the `FileDialog` object set to false, the user can only select one file. The path to this file is returned by the `SelectedItems` property of the `FileDialog` object which is used to load the selected image into an Image control named `imgTest`. You can test this procedure by adding it to the code module of a Worksheet object. Be sure to place an Image control on the worksheet and set its `Name` property before running the program.

```

Public Sub LoadImage()
    Dim fileDiag As FileDialog
    Dim imagePath As String

    Set fileDiag = Application.FileDialog(msoFileDialogFilePicker)
    With fileDiag
        .AllowMultiSelect = False
        .Filters.Clear
    End With

```

```

.Filters.Add Description:=" All files", Extensions:="*.*"
.Filters.Add Description:=" Image", Extensions:="*.bmp",
Position:=1
.FilterIndex = 1
.InitialFileName = ""
.Title = "Select BMP file"
If .Show = -1 Then 'User pressed action button
    imagePath = .SelectedItems(1)
    imgTest.Picture = LoadPicture(imagePath)
End If
End With
End Sub

```

The path to the file selected by the user is returned from the `FileDialog.SelectedItems` collection and stored in the string variable `imagePath`. If the `Execute()` method of the `FileDialog` object is omitted in the program, your program will need this path. Do not use the `Execute()` method of the `FileDialog` object when selecting files that are not compatible with Excel— doing so will either result in a runtime error or open a workbook containing incomprehensible data. If the `AllowMultiSelect` property of the `FileDialog` object is true, the `FileDialog.SelectedItems` collection will hold more than one file path. The `ShowFileDialog()` sub procedure loads the Open dialog box and allows the user to select multiple files. If the user clicks the Open button then the `Execute()` method attempts to open all selected files.

```

Public Sub ShowFileDialog()
    Dim fileDiag As FileDialog
    Const EXCELFILES = 2

    '_____
    'Configure and show the open dialog.
    'Open all files selected by the user.
    '_____

    Set fileDiag = Application.FileDialog(msoFileDialogOpen)
    With fileDiag 'Configure dialog box
        .AllowMultiSelect = True
        .FilterIndex = EXCELFILES
        .Title = "Select Excel File(s)"
        .InitialFileName = ""
        If .Show = -1 Then 'User clicked Open
            .Execute 'Open selected files
        End If
    End With
End Sub

```

The dialog box resulting from the `ShowFileDialog()` sub procedure is shown in Figure 1.35.

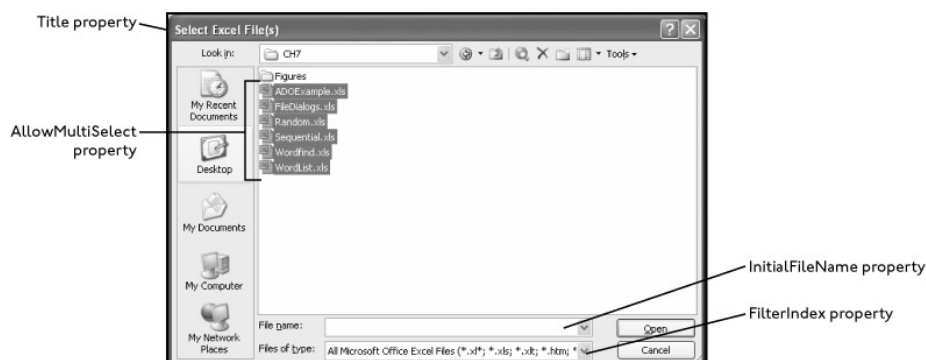


Figure 1.35 - The Open dialog box of the `FileDialog` object.

The FileSystem Object

The FileSystem object is a collection of methods that you can use to set and obtain information about files, directories, and drives. You can find the members of the FileSystem object listed in the Object Browser and in

Table 1.12. You can use them as though they were just another group of VBA built-in functions. That is, you do not need to qualify the object when using these methods in your program.

Table 1.12 - Members Of The File System Object

Member	Description	Example	Return Value
ChDir	Changes the current directory.	ChDir "C:\Documents and Settings" or ChDir ".."	N/A
ChDrive	Changes the current drive.	ChDrive "D:"	N/A
CurDir	Returns the current directory path.	MsgBox CurDir	Outputs the current directory path in a message box.
Dir	Returns the name of a file, directory, or folder that matches a pattern, file attribute, or the volume label of a drive.	fileName = Dir("C:\test.txt", vbNormal)	The file name if it exists. Otherwise an empty string.
EOF	End of file.	EOF(fileNum)	A Boolean value indicating whether the end of an opened file (specified with a file number) has been reached.
FileAttr	The mode used to open a file with the Open statement.	Mode = FileAttr(fileNum, 1)	Returns a Long integer indicating the mode used to open a file (Input, Output, Random, and so on).
FileCopy	Copies a file from a source path to a destination path.	FileCopy "C:\TestFile.txt", "D:\TestFile.txt"	N/A
FileDateTime	Returns the date and time that a file was created or last modified.	fileDate = FileDateTime("C:\test.txt")	
FileLen	Returns the length of a file in bytes.	fileSize = FileLen("C:\test.txt")	
FreeFile	Returns an Integer representing the next file number available for use by the Open statement.	FileNumber = FreeFile	
GetAttr	Returns an Integer representing the attributes of a file or directory.	myAttr = GetAttr(CurDir)	0=Normal, 1=Read-Only, 2=Hidden, 4=System, 16=Directory, 32=Archive
Kill	Deletes a file or files.	Kill "C:\test.txt"	N/A
MkDir	Creates a new directory.	MkDir "TestDir"	N/A
Rmdir	Deletes an empty directory.	Rmdir "TestDir"	N/A
Seek	Returns a Long integer specifying the current read/write position within an open file.	Seek(1)	If the file is opened in Random mode it returns the number of the next record, otherwise it returns the current byte position in the file.

These methods are primarily designed to be used with the Open statement.

1.8.2.4 Sequential Access Files

Writing information to a sequential access file is sort of like recording music to a cassette tape. The songs vary in length and are recorded one after the other. Because it is hard to know the location of each song on the tape, it is difficult to quickly access a particular song. When information is written to a sequential file, the individual pieces of data (usually stored in variables) vary in length and are written to the file one after the other. For example, a sequential file containing names and phone numbers may look something like what's shown here:

"John Smith", "111-2222"

"Joe James", "123-4567"

"Jane Johnson", "456-7890"

The names and phone numbers were all written to the file as strings so they are enclosed in quotes. Numerical values written to a sequential access file will not contain the quotes. The strings containing the names vary in length and will require different amounts of memory for storage. If access to a part of the sequential file is desired at a later time (say we want Jane Johnson's phone number), the entire file must be read into memory because it is not possible to know the location of the desired component within the file. After loading the file, the content must be searched for the desired value. This makes sequential access inefficient with very large files, because it will take too long to access the desired information. However, with smaller files that do not take long to read, sequential access will work well. A VBA sub procedure used to write textual information to a sequential access file is listed here.

```
Public Sub CreateSeqFile()  
    Dim filePath As String  
    Dim I As Integer  
    filePath = ActiveWorkbook.Path & "\SeqPhone.txt"  
    Open filePath For Output As #1  
    For I = 1 To 3  
        Write #1, Cells(I, "A").Value, Cells(I, "B").Value  
    Next I  
    Close #1  
End Sub
```

The preceding procedure uses a For/Next loop to write the contents of the first three cells of columns A and B to a file called SeqPhone.txt. The I/O operation is terminated with the Close statement. The resulting file as viewed from Notepad is shown in Figure 1.32.

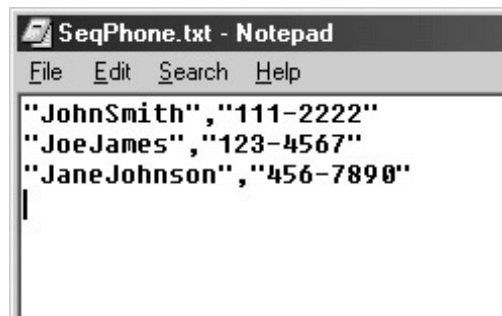


Figure 8.2: Using Notepad to view a sequential file created using VBA code

Using Write # places quotes around each string value written to the file. The file contains three lines of data because Write # adds a new line character to the end of the last value written to the file;

because the For/Next loop iterates three times, the Write # statement was executed three times, resulting in three lines of data. Because the structure of the file is known, it is a simple task to alter the previous procedure to create a new procedure that reads the data.

```
Public Sub ReadSeqFile()
    Dim filePath As String
    Dim I As Integer
    Dim theName As String
    Dim theNumber As String
    I = 1
    filePath = ActiveWorkbook.Path & "\SeqPhone.txt"
    Open filePath For Input As #1
    Do While Not EOF(1)
        Input #1, theName, theNumber
        Cells(I, "A").Value = theName
        Cells(I, "B").Value = theNumber
        I = I + 1
    Loop
    Close #1
End Sub
```

The Open statement in the preceding procedure was changed to allow for data input, and Input # replaces Write #. A Do-Loop replaces the For/Next loop and uses the EOF() function in the conditional. The EOF() function accepts the file number as an argument and returns true when the end of the file is reached. Therefore, the loop continues as long as the EOF() function returns false (Do While NOT False equates to Do While True). Variables must be used to hold the strings returned from the file. Two variables (theName and theNumber) are used to match the structure of the procedure that wrote the data to the file.

1.8.3 More on File Handling

Using files for input and output is a great way for storing data after your program has finished running. If there is information you would like to save until the next time your program runs you can output it to a file. Many other file handling functions like copying or deleting files can be performed in Visual Basic as well. The following contains sample code for doing such operations.

1.8.3.1 Checking for File Existence

```
If Dir$("c:\myfile.txt") = "" then      ' use the Dir command to search
    MsgBox "File not found"
End if
```

1.8.3.2 Listing Files and Folders

This code adds all filenames in a particular directory to a list box

```
Dim sNextFile as String
SNextFile = Dir$("c:\data\*.bmp")

While sNextFile <> ""
    PictureList.AddItem sNextFile
    SNextFile = Dir$
Wend
```

1.8.3.3 Copying Files

```
` copying on a local hard drive
FileCopy "D:\My Documents\MyFile.txt", "C:\Data\MyFile.txt"

` copying from a network drive using variables
Dim sDest as String
Dim sSource as String

sSource = "\\myserver\data\myfile.txt"
sdest = "c:\data\myfile.txt"

FileCopy sSource, sDest
```

1.8.3.4 Deleting Files

```
Kill "d:\data\*.doc"
` this will delete all doc files in the given directory

Kill "d:\myfile.txt"
` this will delete a single file in the given directory
```

1.8.3.5 Renaming Files

```
` use the name command to rename files
Name myoldfile.txt As newfilename.txt

` move files with the name command and make directories
MkDir "d:\newdirectory"
Name "c:\Windows\myfile.txt" As "d:\data\myfile.txt"
```

1.8.3.6 Setting the Current Directory

```
` change the current directory
Chdir "c:\Windows"

` change the current drive letter
Chdrive "C:"
```

1.8.3.7 Sequential File Reading (Input)

To read from a file you first have to **open** the file. Once open, you can read line by line. The example below puts each line from the file into a text box. Remember to always **close** your file when done. It is a good idea to check that the file actually exists before you open it.

```
Dim sTemp as String
` create a string variable to hold a line from our file

Open "c:\data\myfile.txt" For Input as #1
` open a file for input, give it a unique file number
```



```
While Not EOF (1)    ' while we haven't reached the end of file #1

    Line Input #1, sTemp    ' put a line from the file into sTemp
    ListBox.AddItem sTemp    ' put the line into a listbox
Wend

Close #1
```

1.8.3.8 Sequential File Writing (Output)

To output to a file you first have to **open** a file to write to. You have a choice of **output mode** or **append mode**. Appending lets you add text to the end of a file. Once open, you can write line by line to the file. Remember to always **close** your file when done.

1.8.3.9 Output Mode

Output Mode creates a file and allows you to output line by line to it. If you open a file that already exists, the contents of it gets deleted, otherwise your output goes to a new file.

```
Dim fileNumber as Integer
' create a variable to hold our file number

fileNumber = FreeFile    ' this will actually give you a free number

Open "myfile.txt" for Output as #fileNumber
' open a file to write to

Print #fileNumber, "Hello this is a line"
' output a line to the file

Close #fileNumber
' close the file
```

1.8.3.10 Append Mode

Append Mode opens the file specified and allows you to output to the end of it. Whatever you output to this file gets placed at the end of it.

```
Dim fileNumber as Integer
' create a variable to hold our file number

fileNumber = FreeFile    ' this will actually give you a free number

Open "myfile.txt" for Append as #fileNumber
' open a file to write to

Print #fileNumber, "Hello this is a line"
' output a line to the file

Close #fileNumber
' close the file
```

1.8.3.11 Write vs. Print

The **Print** command places a new line marker after it prints out each line. Instead of using the Print command, you can alternatively use the **Write** command. The write command will add delimiters and separators. Other than that, the two are identical.

For example in the file,

```
Write #fileNumber, "Hello this ","is a line"  
Produces -> Hello this is a line
```

```
Print #fileNumber, "Hello this ","is a line"  
Produces -> "Hello this ","is a line"
```