

A Quick Introduction to Programming

A chapter covering the basics of VBScript is the best place to begin this book. This is because of the type of language VBScript is and the kind of users the authors see turning to it. In this chapter, you get a crash course in programming basics. You might not need this chapter because you've come to VBScript with programming skills from another language (Visual Basic, Visual Basic .NET, C, C++, Delphi, C#) and are already both familiar with and comfortable using programming terminology. In that case, feel free to skip this chapter and move on to the next one. However, if you come from a non-programming background, then this chapter will give you the firm foundation you need to begin using VBScript confidently.

If you're still reading, chances are you fall into one of three distinct categories:

- You're a Network/Systems administrator who probably wants to use VBScript and the Windows Script Host or PowerShell to write logon scripts or to automate administration tasks.
- You might be a web designer who feels the need to branch out and increase your skill set, perhaps in order to do some ASP work.
- You're interested in programming (possibly Visual Basic or Visual Basic .NET) and want to check it out before getting too deeply involved.

Programming is a massive subject. Over the years countless volumes have been written about it, both in print and on the Internet. In this chapter, in a single paragraph, we might end up introducing several unfamiliar concepts. We'll be moving pretty fast, but if you read along carefully, trying out your hand at the examples along the way, you'll be just fine.

Also, do bear in mind that there will be a lot that we don't cover here, such as:

- Architecture
- System design
- Database design

- ❑ Documenting code
- ❑ Advanced testing, debugging, and beta testing
- ❑ Rollout and support

Think of this chapter as a brief introduction to the important building blocks of programming. It certainly won't make you an expert programmer overnight, but it will hopefully give you the know-how you'll need to get the most out of the rest of the book.

Variables and Data Types

In this section, you'll quickly move through some of the most basic concepts of programming, in particular:

- ❑ Using variables
- ❑ Using comments
- ❑ Using built-in VBScript functions
- ❑ Understanding syntax issues

Using Variables

Quite simply, a *variable* is a place in the computer memory where your script holds a piece (or pieces) of information, or data. The data stored in a variable can be pretty much anything. It may be something simple, like a small number, like 4, something more complex, like a floating-point number such as 2.3, or a much bigger number like 981.12932134. Or it might not be a number at all and could be a word or a combination of letters and numbers. In fact, a variable can store pretty much anything you want it to store.

Behind the scenes, the variable is a reserved section of the computer's memory for storing data. Memory is temporary — things stored there are not stored permanently like they are when you use the hard drive. Because memory is a temporary storage area, and variables are stored in the computer's memory, they are therefore also temporary. Your script will use variables to store data temporarily that the script needs to keep track of for later use. If your script needs to store that data permanently, it would store it in a file or database on the computer's hard disk.

To make it easier for the computer to keep track of the millions of bits of data that are stored in memory at any given moment, the memory is broken up into chunks. Each chunk is exactly the same size, and is given a unique address. Don't worry about what the memory addresses are or how you use them because you won't need to know any of that to use VBScript, but it is useful to know that a variable is a reserved set of one or more chunks. Also, different types of variables take up different amounts of memory.

In your VBScript program, a variable usually begins its lifecycle by being declared (or dimensioned) before use.

Chapter 1: A Quick Introduction to Programming

It is not required that you declare all of the variables you use. By default, VBScript allows you to use undeclared variables. However, it's strongly recommended that you get into the good habit of declaring all of the variables you use in your scripts. Declaring variables before use makes code easier to read and to debug later. Just do it!

By declaring variables you also give them a name in the process. Here's an example of a variable declaration in VBScript.

```
Dim YourName
```

By doing this, you are in fact giving the computer an instruction to reserve some memory space for you and to name that chunk `YourName`. From now on, the computer (or, more accurately, the VBScript engine) keeps track of that memory for you, and whenever you use the variable name `YourName`, it will know what you're talking about.

Variables are essential to programming. Without them you have no way to hold all the data that your script will be handling. Every input into the script, output from the script, and process within the script uses variables. They are the computer's equivalent of the sticky notes that you leave all over the place with little bits of information on them. All the notes are important (otherwise why write them?) but they are also temporary. Some might become permanent (so you take a phone number and write it down in your address book or contact list), while others are thrown away after use (say, after reminding you to do something). This is how it works with variables, too. Some hold data that you might later want to keep, while others are just used for general housekeeping and are disposed of as soon as they're used.

In VBScript, whenever you have a piece of information that you need to work with, you declare a variable using the exact same syntax you saw a moment ago. At some point in your script, you'll need to do something with the memory space you've allocated yourself (otherwise, what would be the point of declaring it?). And what you do with a variable is place a value in it. This is called *initializing* the variable. Sometimes you initialize a variable with a default value. Other times, you might ask the user for some information, and initialize the variable with whatever the user enters. Alternatively, you might open a database and use a previously stored value to initialize the variable.

When we say database, we don't necessarily mean an actual database but any store of data — it might be an Internet browser cookie or a text file that we get the data from. If you are dealing with small amounts of data a cookie or text file will suffice, but if you are dealing with a lot of data you need the performance and structure that a database offers.

Initializing the variable gives you a starting point. After it has been initialized, you can begin making use of the variable in your script.

Here's a very simple VBScript example.

```
Dim YourName
' Above we dimensioned the variable
YourName = InputBox("Hello! What is your name?")
' Above we ask for the user's name and initialize the variable
MsgBox "Hello " & YourName & "! Pleased to meet you."
' Above we display a greeting containing the user's name
```

Rightly so, you're now probably wondering what all this code means. Last time, you were showed one line and now it's grown to six.

Chapter 1: A Quick Introduction to Programming

All of the examples in this chapter are designed so that you can run them using the Windows Script Host (WSH). The WSH is a scripting host that allows you to run VBScript programs within Windows. WSH allows you to try out these example programs for yourself. You may already have WSH installed. To find out, type the previous example script into a text editor, save the file as TEST.VBS (it must have the .VBS extension, and not a .TXT), and double-click the file in Windows Explorer. If the script runs, then you're all set. If Windows does not recognize the file, then you need to download and install WSH from <http://msdn2.microsoft.com/en-us/library/ms950396.aspx>.

Using Comments

You already know what the first line of code in the previous block does. It declares a variable for use called `YourName`.

The second line in the code is a comment. In VBScript, any text preceded by the single quote character (`'`) is treated as a comment, which means that the VBScript engine completely ignores the text, which begs the question why bother typing it in at all? It doesn't contribute to the execution of the script, right? This is absolutely correct, but don't forget one of the most important principles of programming: It is not just computers that may have to read script. It is equally important to write a script with human readers in mind as it is to write with the computer in mind.

Of course, none of this means you should for one moment forget that when you write scripts, you must do so with the computer (or, more specifically, the script engine) in mind. If you don't type the code correctly (that is, if you don't use the proper syntax), the script engine won't be able to execute the script. However, once you've written some useful scripts, you'll probably need to go back to make some changes to a script you wrote six months or a year ago. If you didn't write that code with human readers, as well as computers, in mind it could be pretty difficult to figure out what you were thinking and how you decided to solve the problems at the time you wrote the script. Things can get worse. What happens when you or one of your coworkers has to make some changes to a script you wrote many months ago? If you did not write that script to be both readable and maintainable, others who use your code will encounter difficulties deciphering it — no matter how well written the actual computer part of the code is.

Adding comments to your code is just one part of making sure code is clear and readable. There are many other things that you can do:

- Choose clear, meaningful variable names.
- Indent code for clarity.
- Make effective use of white space.
- Organize the code in a logical manner.

All of these aid human-readability and are covered later, but clear, concise comments are by far the most important. However, too much of a good thing is never good and the same is true for comments. Overburdening code with comments doesn't help. Remember that if you are scripting for the Web that all the code, including the comments, are downloaded to the browser, so unnecessary comments may adversely affect download times.

You learn about some good commenting principles later in this chapter, but for now just be aware of the fact that the comment in line 2 of the script is not really a good comment for everyday use. This is because, to any semi-experienced programmer, it is all too obvious that what you are doing is declaring

the `YourName` variable on the code line above. However, throughout this book you'll often see the code commented in a similar way. This is because the point of the code is to instruct the reader in how a particular aspect of VBScript programming works, and the best way to do that is to add comments to the code directly. It removes ambiguity and keeps the code and comments together.

Also worth noting is that comments don't have to be on a separate line. Comments can also follow the code, like so:

```
Dim YourName ' initialize the variable
YourName = InputBox("Hello! What is your name?") ' ask for the user's name
MsgBox "Hello " & YourName & "! Pleased to meet you." ' display a greeting
```

This works in theory but it isn't as clear as keeping the comments on separate lines in the script.

Using Built-in VBScript Functions

OK, back to the script. Take a look at line 3.

```
YourName = InputBox("Hello! What is your name?")
```

Here you are doing two things at once. First, you're initializing the variable. You could do it directly, like this:

```
YourName = "Fred"
```

However, the drawback with this is that you're making the arbitrary decision that everyone is called `Fred`, which is ideal for some applications but not for others. If you wanted to assign a fixed value to a variable, such as a tax rate, this would be fine.

```
Dim TaxRate
TaxRate = 17.5
```

Because you want to do something that gives the user a choice, you should employ the use of a function, called `InputBox`. This function and all the others are discussed in later chapters, but for now all you need to know is that `InputBox` is used to display a message in a dialog box, and it waits for the user to input text or click a button. The `InputBox` generated is displayed in Figure 1-1.

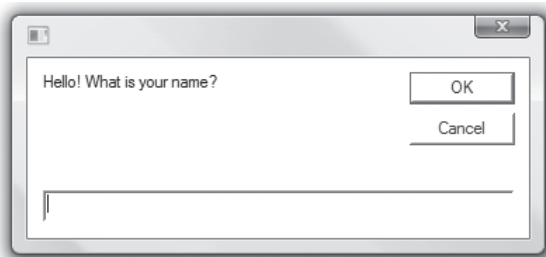


Figure 1-1

The clever bit is what happens to the text that the user types into the input box displayed — it is stored in the variable `YourName`.

Chapter 1: A Quick Introduction to Programming

Line 4 is another comment. Line 5 is more code. Now that you've initialized this variable, you're going to do something useful with it. `MsgBox` is another built-in VBScript function that you will probably use a lot during the course of your VBScript programming. Using the `MsgBox` function is a good way to introduce the programming concept of passing function parameters, also known as *arguments*. Some functions don't require you to pass parameters to them while others do. This is because some functions (take the `Date` function as an example — this returns the current date based on the system time) do not need any additional information from you in order to do their job. The `MsgBox` function, on the other hand, displays a piece of information to the user in the form of a dialog box, such as the one shown in Figure 1-2.

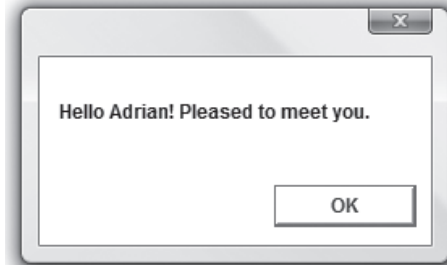


Figure 1-2

You have to pass `MsgBox` a parameter because on its own it doesn't have anything useful to display (in fact, it will just bring up a blank pop-up box). The `MsgBox` function actually has several parameters, but for now you're just going to look at one. All of the other parameters are optional parameters.

Understanding Syntax Issues

Take another look at line 5 and you'll probably notice the ampersand (&). The ampersand is a VBScript operator, and is used to concatenate (join) pieces of text together. To concatenate simply means to "string together." This text can take the form of either a literal or a variable. A literal is the opposite of a variable. A variable is so named because it is exactly that — a variable — and can change throughout the lifetime of the script (a script's lifetime is the time from when it starts executing, to the time it stops). Unlike a variable, a literal cannot change during the lifetime of the script. Here is line 5 of the script again.

```
MsgBox "Hello "& YourName & "! Pleased to meet you."
```

An operator is a symbol or a word that you use within your code that is usually used to change or test a value. Other operators include the standard mathematical operators (+, -, /, *), and the equals sign (=), which can actually be used in either a comparison or an assignment. So far, you've used the equals sign as an assignment operator. Later in this chapter you'll find out more about operators.

Now take a closer look at variables. Remember how we said that a variable is a piece of reserved memory? One question you might have is, How does the computer know how large to make that piece of memory? Well, again, in VBScript this isn't something that you need to worry about and it is all handled automatically by the VBScript engine. You don't have to worry in advance about how big or small you need to make a variable. You can even change your mind and the VBScript engine will dynamically change and reallocate the actual memory addresses that are used up by a variable. For example, take a quick look at this VBScript program.

```
' First declare the variable
Dim SomeVariable

' Initialize it with a value
SomeVariable = "Hello, World!"
MsgBox SomeVariable

' Change the value of the variable to something larger
SomeVariable = "Let's take up more memory than the previous text"
MsgBox SomeVariable

' Change the value again
SomeVariable = "Bye!"
MsgBox SomeVariable
```

Each time the script engine comes across a variable, the engine assigns it the smallest chunk of memory it needs. Initially the variable contains nothing at all so needs little space but as you initialize it with the string "Hello, World!" the VBScript engine asks the computer for more memory to store the text. But again it asks for just what it needs and no more. (Memory is a precious thing and not to be wasted.) Next, when you assign more text to the same variable, the script engine must allocate even more memory, which it again does automatically. Finally, when you assign the shorter string of text, the script engine reduces the size of the variable in memory to conserve memory.

One final note about variables: Once you've assigned a value to a variable, you don't have to throw it away in order to assign something else to the variable as well. Take a look at this example.

```
Dim SomeVariable

SomeVariable = "Hello"
MsgBox SomeVariable

SomeVariable = SomeVariable & ", World!"
MsgBox SomeVariable

SomeVariable = SomeVariable & " Goodbye!"
MsgBox SomeVariable
```

Notice how in this script, you each time keep adding the original value of the variable and adding some additional text to it. You tell the script engine that this is what you want to do by also using the name of the `SomeVariable` variable on the right side of the equals sign, and then concatenating its existing value with an additional value using the ampersand (&) operator. Adding onto the original value works with numbers, too (as opposed to numbers in strings) but you have to use the + operator instead of the & operator.

```
Dim SomeNumber

SomeNumber = 999
MsgBox SomeNumber

SomeNumber = SomeNumber + 2
MsgBox SomeNumber

SomeNumber = SomeNumber + 999
MsgBox SomeNumber
```

Chapter 1: A Quick Introduction to Programming

Here are the resulting message boxes generated by this code. The first is shown in Figure 1-3.



Figure 1-3

The second message box is shown in Figure 1-4.



Figure 1-4

The final message box is shown in Figure 1-5.



Figure 1-5

You can store several different types of data in variables. These are called data types and so far you've seen two:

- ❑ String
- ❑ Integer

You've also seen a single-precision floating-point number in the tax rate example.

We'll be covering all of them later on in the book. For now, just be aware that there are different data types and that they can be stored in variables.

Flow Control

When you run a script that you have written, the code executes in a certain order. This order of execution is also known as *flow*. In simple scripts such as the ones you looked at so far, the statements simply execute from the top down. The script engine starts with the first statement in the script, executes it, moves on to the next one, and then the next one, and so on until the script reaches the end. The execution occurs this way because the simple programs you've written so far do not contain any branching or looping code.

Branching

Take a look at a script that was used earlier.

```
Dim YourName
'Above we initialized the variable
YourName = InputBox("Hello! What is your name?")
'Above we ask for the user's name and initialize the variable
MsgBox "Hello " & YourName & "! Pleased to meet you."
'Above we display a greeting containing the user's name
```

If you save this script in a file with a `.vbs` extension, and then execute it using the Windows Script Host, all of the statements will be executed in order from the first statement to the last.

Note that it was previously mentioned that all of the statements will be executed. However, this isn't what you always want. There is a technique that you can use to cause some statements to be executed, and some not, depending on certain conditions. This technique is called *branching*.

VBScript supports a few different branching constructs, and they are covered in detail in Chapter 5, but here we only cover the simplest and most common one, which is the `If...Else...End If` construct.

Take a look at this modified code example.

Chapter 1: A Quick Introduction to Programming

```
Dim YourName
Dim Greeting

YourName = InputBox("Hello! What is your name?")

If YourName = "" Then
    Greeting = "OK. You don't want to tell me your name."
Else
    Greeting = "Hello, "& YourName & ", great to meet you."
End If

MsgBox Greeting
```

Walking through the code, you do the following:

1. You declare the two variables that you are going to use:

```
Dim YourName
Dim Greeting

YourName = InputBox("Hello! What is your name?")
```

You ask the user for some input, again using the `InputBox` function. This function expects one required parameter, the prompt text (the text that appears on the input box). It can also accept several optional parameters. Here, you only use the one required parameter.

Note that the parameter text that you passed "Hello! What is your name?" is displayed as a prompt for the dialog box. The `InputBox` function returns the value that the user types, if any. If the user does not type anything or clicks the `Cancel` button (both do the same thing), then `InputBox` returns a zero-length string, which is a strange kind of programming concept that basically means that it returns text that doesn't actually contain any text. Your script stores the result of the `InputBox` function in the `YourName` variable.

2. You come to the actual loop you're going to use:

```
If YourName = "" Then
    Greeting = "OK. You don't want to tell me your name."
Else
    Greeting = "Hello, "& YourName & ", great to meet you."
End If
```

This code presents the VBScript engine with an option that is based on what the user typed (or didn't type) into the input box. The first line tests the input from the user. It tests to see if the input that is stored in the variable `YourName` is a zero-length string. If it is, the next line of code is run and the variable `Greeting` is assigned a string. Figure 1-6 shows the message displayed if the user doesn't type his or her name into the `InputBox`.

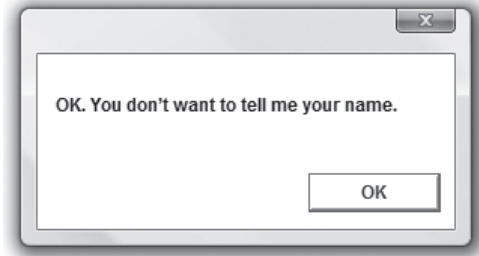


Figure 1-6

3. What happens if the user does (as you expect) type something into the input box? Well, this is where the next line comes in.

```
Else
```

You can actually begin to read the code and in fact doing this helps it to make sense. What the whole loop actually means is that if the value of variable `YourName` is a zero-length string, then assign the variable `Greeting` with one value; however, if it contains something else, do something else (assign `Greeting` a different value). This doesn't protect your script from users entering data like numbers or non-alphabet characters into the test box, although you could code for all these conditions if you wanted to.

4. The final line of the code uses the `MsgBox` function to display the value of the variable `Greeting`. Notice that both lines of code assign a value to the `Greeting` variable. However, only one of these lines will actually execute in any one running of the script. This is because the `If...Else...End If` block makes an either/or decision. Either a given condition is `True`, or it is `False`. There's no way it can be neither (not a string that contains text nor a zero-length string) or both (a zero-length string that contains text). If it is `True`, then the script engine will execute the code between the `If` and `Else` statements. If it is `False`, then it will execute the code between the `Else` and `End If` statements.

So, what the complete script does is test the input, and then executes different code, depending on the result of that test, and hence the term branching. Using this technique allows your script to adapt to the unpredictable nature of the input. Compare the intelligent script to the following one, which looks pretty lame.

```
Dim YourName
Dim Greeting
YourName = InputBox("Hello! What is your name?")

Greeting = "Hello, "& YourName & ", great to meet you."

MsgBox Greeting
```

This script is just plain dumb because it does not contain any branching logic to test the input; so when the user does something unpredictable, such as clicking the `Cancel` button, or not entering any name at all, the script does not have the ability to adapt. Compare this to your intelligent script, which is capable of adapting to the unpredictability of input by testing it with `If...Else...End If` branching.

Chapter 1: A Quick Introduction to Programming

Before you move on to looping, you should know a few other things about `If...Else...End If`:

- ❑ The block of code containing the `If...Else...End If` is known as a block of code. A block is a section of code that has a beginning and an end, and it usually contains keywords or statements at both the beginning and the end. In the case of `If...Else...End If`, the `If` statement marks the beginning of the block, while the `End If` marks the end of the block.

The script engine requires these beginning and ending statements, and if you omit them, the script engine won't understand your code and won't allow your script to execute. Over the course of this book you will encounter many different types of code blocks in VBScript.

To confuse matters, the term "block of code" is often used informally to describe any group of lines of code. As a rule, "block of code" will refer to lines of code that work together to achieve a result.

- ❑ Notice that the lines of code that are inside the block itself are indented by four spaces. This is an extremely important concept but not for the reason you might think. This indenting has nothing whatsoever to do with the script engine — it doesn't care whether you add four spaces, 44 spaces, or none at all. This indenting is for the benefit of any humans who might be reading your code. For example, the following script is completely legal and will execute just fine:

```
Dim YourName

    Dim Greeting

    YourName = InputBox("Hello! What is your name?")

    If YourName = "" Then

        Greeting = "OK. You don't want to tell me your name."
    Else

        Greeting = "Hello, "& YourName & ", great to meet you."
    End If

    MsgBox Greeting
```

However, this code is very difficult to read. As a general rule of thumb, you indent code by four spaces whenever a line or series of lines is subordinate to the lines above and below it. For example, the lines after the `If` clause and the `Else` clause belong inside the `If...Else...End If` block, so you indent them to visually suggest the code's logical structure. Presentation, while having no bearing whatsoever on how the computer or script engine handles your code, is very important when it comes to how humans read it. You should be able to look at the code and get a sense for how it is organized and how it works. By seeing the indentations inside the `If...Else...End If` block, you can not only read the code, but also "see" the branching logic at that point in the code. Indenting is only one element of programming style, but learning and following proper style and layout is essential for any programmer who wants to be taken seriously.

- ❑ The `Else` part of the block is optional. Sometimes you want to test for a certain condition, and if that condition is `True`, execute some code, but if it's `False`, there's no code to execute. For example, you could add another `If...End If` block to your script.

```
Dim YourName
Dim Greeting

YourName = InputBox("Hello! What is your name?")

If YourName = "" Then
    Greeting = "OK. You don't want to tell me your name."
Else
    Greeting = "Hello, " & YourName & ", great to meet you."
End If
```

```
If YourName = "Fred" Then
    Greeting = Greeting & " Nice to see you Fred."
End If
```

```
MsgBox Greeting
```

- The `If...Else...End If` block can be extended through the use of the `ElseIf` clause, and through nesting. *Nesting* is the technique of placing a block of code inside of another block of code of the same type. The following variation on your script illustrates both concepts:

```
Dim YourName
Dim Greeting

YourName = InputBox("Hello! What is your name?")

If YourName = "" Then
    Greeting = "OK. You don't want to tell me your name."

ElseIf YourName = "abc" Then
    Greeting = "That's not a real name."
ElseIf YourName = "xxx" Then
    Greeting = "That's not a real name."
Else
    Greeting = "Hello, "& YourName & ", great to meet you."

    If YourName = "Fred" Then
        Greeting = Greeting & " Nice to see you Fred."
    End If

End If

MsgBox Greeting
```

Once again, seeing how the code has been indented helps you to identify which lines of code are subordinate to the lines above them. As code gets more and more complex, proper indenting of the code becomes vital as it will become harder to follow.

- Even though the branching logic you are adding to the code tells the script to execute certain lines of code while not executing others, all the code must still be interpreted by the script engine (including the code that's not executed). If any of the code that's not executed contains any syntax errors, the script engine will still produce an error message to let you know.

Looping

Branching allows you to tell the script to execute some lines of code, but not others. *Looping*, on the other hand, allows you to tell the script to execute some lines of code over and over again. This is particularly useful in two situations:

- ❑ When you want to repeat a block of code until a condition is `True` or `False`
- ❑ When you want to repeat a block of code a finite number of times

There are many different looping constructs, but this section focuses on only two of them:

- ❑ The basic `Do...Loop While` loop
- ❑ The basic `For...Next` loop

Using the `Do...Loop While` Loop

This section takes a look at the `Do...Loop While` construct and how it can be used to repeatedly execute a block of code until a certain condition is met. Take a look at the following modification of the example script:

```
Dim Greeting
Dim YourName
Dim TryAgain

Do
    TryAgain = "No"

    YourName = InputBox("Please enter your name:")

    If YourName = "" Then
        MsgBox "You must enter your name to continue."
        TryAgain = "Yes"
    Else
        Greeting = "Hello, "& YourName & ", great to meet you."
    End If

Loop While TryAgain = "Yes"

MsgBox Greeting
```

Notice the block of code that starts with the word `Do` and ends with the line that starts with the word `Loop`. The indentation should make this code block easy to identify. This is the definition of the loop. The code inside the loop will keep being executed until at the end of the loop the `TryAgain` variable equals `"No"`.

The `TryAgain` variable controls the loop. The loop starts at the word `Do`. At the end of the loop, if the `TryAgain` variable equals `"Yes"`, then all the code, starting at the word `Do`, will execute again.

Notice that the top of the loop initializes the `TryAgain` variable to `"No"`. It is absolutely essential that this initialization take place inside the loop (that is, between the `Do` and `Loop` statements). This way, the variable is reinitialized every time a loop occurs. If you didn't do this, you would end up with what's called an infinite loop. They are always bad. At best, the user is going to have to exit out of the program in an untimely (and inelegant) way because, as the name suggests, the loop is infinite. At worse, it can crash the system. You want neither and you want to try to avoid both in your code.

Take a look at why the `TryAgain = "No"` line is essential to preventing an infinite loop. Going through the script line by line:

1. This first line starts the loop.

Do

This tells the script engine that you are starting a block of code that will define a loop. The script engine will expect to find a loop statement somewhere further down in the script. This is similar to the `If...End If` code block because the script engine expects the block to be defined with beginning and ending statements. The `Do` statement on a line all by itself means that the loop will execute at least once. Even if the `Loop While` statement at the end of the block does not result in a loop around back to the `Do` line, the code inside this block will be executed at least one time.

2. Moving on to the second line of code, you initialize the “control” variable. It’s called the “control” variable because it ultimately controls whether or not the code block loops around again. You want to initialize this variable to “No” so that, by default, the loop will not loop around again. Only if a certain condition is met inside the loop will you set `TryAgain` to “Yes”. This is yet another strategy in an ever-vigilant desire to expect the unexpected.

Do

```
TryAgain = "No"
```

3. The next line of code should look familiar. You use the `InputBox` function to ask the user to enter a name. You store the return value from the function in the `YourName` variable. Whatever the user types, unless they type nothing, will be stored in this variable. Put another way, the script receives some external input — and remember that we said input is always unpredictable:

Do

```
TryAgain = "No"
```

```
YourName = InputBox("Please enter your name:")
```

4. In the next part of the code, you test the input. The line `If YourName = ""` Then tests to see if the user typed in their name (or at least some text). If they typed something in, the code immediately after the `Else` line will execute. If they didn’t type in anything (or if they clicked the `Cancel` button), then the `YourName` variable will be empty, and the code after the `If` line will execute instead:

Do

```
TryAgain = "No"
```

```
YourName = InputBox("Please enter your name:")
```

```
If YourName = "" Then
    MsgBox "You must enter your name to continue."
    TryAgain = "Yes"
Else
    Greeting = "Hello, "& YourName & ", great to meet you."
End If
```

If the user didn’t type anything into the input box, you will display a message informing them that they have done something you didn’t want them to. You then set the `TryAgain` variable (the control variable) to “Yes” and send them around the loop once more and ask the users

Chapter 1: A Quick Introduction to Programming

for their name again (wherein this time they will hopefully type something into the input box). If the user did type in his or her name, then you initialize your familiar `Greeting` variable. Note that in this case, you do not change the value of the `TryAgain` variable. This is because there is no need to loop around again because the user has entered a name. The value of `TryAgain` is already equal to "No", so there's no need to change it.

5. In the next line of code, you encounter the end of the loop block. What this `Loop` line is essentially telling the script engine is "If the `TryAgain` variable equals "Yes" at this point, then go back up to the `Do` line and execute all that code over again." If the user entered his or her name, then the `TryAgain` variable will be equal to "No". Therefore, the code will not loop again, and will continue onto the last line:

```
Do
    TryAgain = "No"

    YourName = InputBox("Please enter your name:")

    If YourName = "" Then
        MsgBox "You must enter your name to continue."
        TryAgain = "Yes"
    Else
        Greeting = "Hello, "& YourName & ", great to meet you."
    End If
```

```
Loop While TryAgain = "Yes"
```

```
MsgBox Greeting
MsgBox Greeting
```

If the user did not enter his or her name, then `TryAgain` would be equal to "Yes", which would mean that the code would again jump back to the `Do` line. This is where the reinitialization of the `TryAgain` variable to "No" is essential because if it wasn't done then there's no way for `TryAgain` to ever equal anything but "Yes". And if `TryAgain` always equals "Yes", then the loop will keep going around and around forever. This results in total disaster for your script, and for the user.

Using the For...Next Loop

In this kind of loop, you don't need to worry about infinite loops because the loop is predefined to execute only a certain number of times. Here's a simple (if not very useful) example.

```
Dim Counter

MsgBox "Let's count to ten. Ready?"

For Counter = 1 to 10
    MsgBox Counter
Next

MsgBox "Wasn't that fun?"
```

This loop is similar to the previous loop. The beginning loop block is defined by the `For` statement, and the end is defined by the `Next` statement. This loop is different because you can predetermine how many times it will run; in this case, it will go around exactly ten times. The line `For Counter = 1 to 10` essentially tells the script engine, "Execute this block of code as many times as it takes to count from

Chapter 1: A Quick Introduction to Programming

1 to 10, and use the `Counter` variable to keep track of your counting. When you've gone through this loop ten times, stop looping and move on to the next bit of code."

Notice that every time the loop goes around (including the first time through), the `Counter` variable holds the value of the current count. The first time through, `Counter` equals 1, the second time through it equals 2, and so on up to 10. It's important to note that after the loop is finished, the value of the `Counter` variable will be 11, one number higher than the highest value in your `For` statement. The reason for this is that the `Counter` variable is incremented at the end of the loop, after which the `For` statement tests the value of `index` to see if it is necessary to loop again.

Giving you a meaningful example of how to make use of the `For . . . Next` loop isn't easy because you haven't been exposed to much VBScript just yet, but here's an example that shows you don't need to know how many times the loop needs to run before you run it.

```
Dim Counter
Dim WordLength
Dim WordBuilder

WordLength = Len("VBScript is great!")

For Counter = 1 to WordLength
    MsgBox Mid("VBScript is great!", Counter, 1)
    WordBuilder = WordBuilder & Mid("VBScript is great!", Counter, 1)
Next

MsgBox WordBuilder
```

For example, the phrase "VBScript is great!" has exactly 18 letter spaces. If you first calculated the number of letters in the phrase, you could use that number to drive a `For . . . Next` loop. However, this code uses the VBScript `Len()` function to calculate the length of the phrase used. Inside the loop, it uses the `Mid()` function to pull one letter out of the phrase one at a time and display them separately. The position of that letter is controlled by the counter variable, while the number of letters extracted is defined by the `length` argument at the end. It also populates the `WordBuilder` variable with each loop, adding each new letter to the previous letter or letters, rebuilding the phrase.

Here's a variation of the last example: here giving the user the opportunity to type in a word or phrase to use, proving that there's nothing up your sleeve when it comes to knowing how many times to loop the code.

```
Dim Counter
Dim WordLength
Dim InputWord
Dim WordBuilder

InputWord = InputBox ("Type in a word or phrase to use")

WordLength = Len(InputWord)

For Counter = 1 to WordLength
    MsgBox Mid(InputWord, Counter, 1)
    WordBuilder = WordBuilder & Mid(InputWord, Counter, 1)
Next

MsgBox WordBuilder & " contains "& WordLength & " characters."
```

Figure 1-7 shows the final summary message generated by the code. Notice how well the information is integrated.

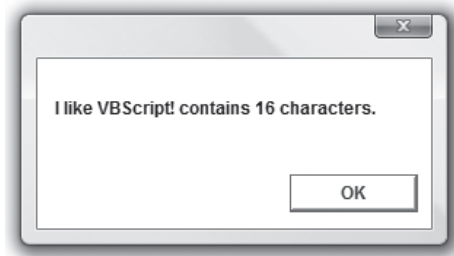


Figure 1-7

Operators and Operator Precedence

An operator acts on one or more operands when comparing, assigning, concatenating, calculating, and performing logical operations. Say you want to calculate the difference between two variables x and y and save the result in variable z . These variables are the operands and to find the difference you use the subtraction operator like this:

```
Z = X - Y
```

Here you use the assignment operator (=) to assign the difference between x and y , which was found by using the subtraction operator (-).

Operators are one of the single-most important parts of any programming language. Without them, you cannot assign values to variables or perform calculations or comparisons. In fact, you can't do much at all.

There are different types of operators and they each serve a specific purpose, as shown in the following table.

Operator	Purpose
assignment (=)	The most obvious and is simply used for assigning a value to a variable or property.
arithmetic	These are all used to calculate a numeric value, and are normally used in conjunction with the assignment operator and/or one of the comparison operators.
concatenation	These are used to concatenate ("join together") two or more different expressions.
comparison	These are used for comparing variables and expressions against other variables, constants, or expressions.
logical	These are used for performing logical operations on expressions; all logical operators can also be used as bitwise operators.
bitwise	These are used for comparing binary values bit by bit; all bitwise operators can also be used as logical operators.

When you have a situation where more than one operation occurs in an expression, the operations are normally performed from left to right. However, there are several rules.

Operators from the arithmetic group are evaluated first, then concatenation, comparison, and finally logical operators. This is the set order in which operations occur (operators in brackets have the same precedence):

- ❑ $\cap, -, (*, /), \setminus, \text{Mod}, (+, -)$
- ❑ $\&$
- ❑ $=, <>, <, >, <=, >=, \text{Is}$
- ❑ $\text{Not}, \text{And}, \text{Or}, \text{Xor}, \text{Eqv}, \text{Imp}$

This order can be overridden by using parentheses. Operations in parentheses are evaluated before operations outside the parentheses, but inside the parentheses, the normal precedence rules still apply.

Take a look at the following two statements:

```
A = 5 + 6 * 7 + 8
A = (5 + 6) * (7 + 8)
```

They look the same but they're not. According to operator precedence, multiplication is performed before addition, so the top line gives A the value 55 ($6 * 7 = 42 + 5 + 8 = 55$). By adding parentheses, you force the additions to be evaluated first and A becomes equal to 165.

Organizing and Reusing Code

So far, the scripts you've worked with have been fairly simple in structure. The code has been all together in one unit. You haven't done anything all that complicated, so it's easy to see all the code in just a few lines. The execution of the code is easy to follow because it starts at the top of the file, with the first line, and then continues downward until it reaches the last line. Sometimes, at certain points, choices redirect the code using branching, or sections of code are repeated using loops.

However, when you come to writing a script that actually does something useful, your code is likely to get more complex. As you add more code to the script, it becomes harder to read in one chunk. If you print it on paper, your scripts will undoubtedly stretch across multiple pages. As the code becomes more complex, it's easier for bugs and errors to creep in, and the poor layout of the code will make these harder to find and fix. The most common technique programmers use to manage complexity is called *modularization*. This is a big, fancy word, but the concept behind it is really quite simple.

This section defines some terminology used when organizing and reusing code, and then discusses how to write your own procedures by turning code into a function. You then learn a few advantages of having procedures.

Modularization, Black Boxes, Procedures, and Subprocedures

Modularization is the process of organizing your code into modules, which you can also think of as building blocks. You can apply the principles of modularity to create your own personal set of programming building blocks, which you can then use to build programs that are more powerful, more reliable, easier to debug, and easier for you and your fellow programmers to maintain and reuse. When you take your code and divide it into modules, your ultimate goal is to create what are known as black boxes. A *black box* is any kind of device that has a simple, well-defined interface and that performs some discrete, well-defined function. A black box is so called because you don't need to see what's going on inside it. All you need to know is what it does, what its inputs are, and (sometimes) what its outputs are.

A wristwatch is a good example of a black box. It has inputs (buttons) and outputs (time) and does a simple function well without you worrying about how the innards of the watch work in order to be able to tell the time. The most basic kind of black box programmers use to achieve modularity is the procedure. A *procedure* is a set of code that (ideally) performs a single function. Good examples of procedures are:

- Code that adds two numbers together
- Code that processes a string input
- Code that handles saving to a file

Bad examples include:

- Code that takes an input, processes it, and also handles saving to a file
- Code that handles file access and database access

You've been using procedures throughout this chapter, but they have been procedures that VBScript provides for you. Some of these procedures require input, some don't. Some of these procedures return a value, some don't. But all of the procedures you've used so far (`MsgBox()`, `InputBox()`, and so on) are black boxes. They perform one single well-defined function, and they perform it without you having to worry about how they perform their respective functions. In just a moment, you'll see how to extend the VBScript language by writing your own procedures.

Before you begin though, it's time to get some of the terminology cleared up. *Procedure* is a generic term that describes either a function or a subprocedure. This chapter touched on some of this confusing terminology earlier, but a *function* is simply a procedure that returns a value. `Len()` is a function. You pass it some text, and it returns the number of characters in the string (or the number of bytes required to store a variable) back to you. Functions do not always require input, but they often do.

A *subprocedure* is a procedure that does not return a value. You've been using `MsgBox()` as a subprocedure. You pass it some text, and it displays a message on the screen comprising of that text. It does not return any kind of value to your code. All you need to know is that it did what you asked it to do. Just like functions, procedure may or may not require input.

Turning Code into a Function

Some of the code that follows is from an example you used earlier in the chapter. Here's how to turn code into a function.

```
Function PromptUserName

' This Function prompts the user for his or her name.
' If the user enters nothing it returns a zero-length string.
' It incorporates various greetings depending on input by the user.
Dim YourName
Dim Greeting

YourName = InputBox("Hello! What is your name?")

If YourName = "" Then
    Greeting = "OK. You don't want to tell me your name."
ElseIf YourName = "abc" Then
    Greeting = "That's not a real name."
ElseIf YourName = "xxx" Then
    Greeting = "That's not a real name."
Else
    Greeting = "Hello, " & YourName & ", great to meet you."

    If YourName = "Fred" Then
        Greeting = Greeting & " Nice to see you Fred."
    End If

End If

MsgBox Greeting

PromptUserName = YourName

End Function
```

The first things to take note of in the code are the first and last lines. While not groundbreaking, these are what define a function. The first line defines the beginning of the function and gives it a name while the last line defines the end of the function. Based on the earlier discussion of code blocks, this should be a familiar convention by now. From this, you should begin to realize that a procedure is nothing but a special kind of code block. The code has to tell the script engine where it begins and where it ends. Notice also that you've given the function a clear, useful name that precisely describes what this function does. Giving your procedures good names is one of the keys to writing programs that are easy to read and maintain.

Notice also how there's a comment to the beginning of the procedure to describe only what it does, not how the function does what it does. The code that uses this function does not care how the function accomplishes its task; it only cares about inputs, outputs, and predictability. It is vitally important that you add clear, informative comments such as this to the beginning of your procedures, because they make it easy to determine what the function does. The comment also performs one other valuable service to you and any other developer who wants to call this function — it says that the function may return a zero-length string if the user does not enter his or her name.

Chapter 1: A Quick Introduction to Programming

Finally, notice how, in the second to last line, the function name `PromptUserName` is treated as if it were a variable. When you use functions (as opposed to subprocedures, which do not return a value), this is how you give the function its return value. In a sense, the function name itself is a variable within the procedure.

Here is some code that uses the `PromptUserName` function.

```
Dim Greeting
Dim VisitorName

VisitorName = PromptUserName

If VisitorName <> "" Then
    Greeting = "Goodbye, " & VisitorName & ". Nice to have met you."
Else
    Greeting = "I'm glad to have met you, but I wish I knew your name."
End If

MsgBox Greeting
```

If you are using Windows Script Host for this code, bear in mind that this code and the `PromptUserName` function itself must be in the same `.vbs` script file.

```
Dim PartingGreeting
Dim VisitorName

VisitorName = PromptUserName

If VisitorName <> "" Then
    PartingGreeting = "Goodbye, " & VisitorName & ". Nice to have met you."
Else
    PartingGreeting = "I'm glad to have met you, but I wish I knew your name."
End If

MsgBox PartingGreeting

Function PromptUserName

    ' This Function prompts the user for his or her name.
    ' It incorporates various greetings depending on input by the user.
    Dim YourName
    Dim Greeting

    YourName = InputBox("Hello! What is your name?")

    If YourName = "" Then
        Greeting = "OK. You don't want to tell me your name."
    ElseIf YourName = "abc" Then
        Greeting = "That's not a real name."
    ElseIf YourName = "xxx" Then
        Greeting = "That's not a real name."
    End If
End Function
```

```
Else
    Greeting = "Hello, " & YourName & ", great to meet you."

    If YourName = "Fred" Then
        Greeting = Greeting & " Nice to see you Fred."
    End If

End If

MsgBox Greeting

PromptUserName = YourName

End Function
```

As you can see, calling the `PromptUserName` function is pretty straightforward. Once you have written a procedure, calling it is no different than calling a built-in VBScript procedure.

Advantages to Using Procedures

Procedures afford several key advantages that are beyond the scope of this discussion. However, here are a few of the most important ones:

- ❑ Code such as that put in the `PromptUserName` function can be thought of as “generic,” meaning that it can be applied to a variety of uses. Once you have created a discreet, well-defined, generic function such as `PromptUserName`, you are free to reuse it any time you want to prompt users for their name. Once you’ve written a well-tested procedure, you never have to write that code again. Any time you need it, you just call the procedure. This is known as code reuse.
- ❑ When you call a procedure to perform a task rather than writing the code in-line, it makes that code much easier to read and maintain. Increasing the readability, and therefore the manageability and maintainability, of your code is a good enough reason to break a block of code out into its own procedure.
- ❑ When code is isolated into its own procedure, it greatly reduces the effects of changes to that code. This goes back to the idea of the black box. As long as the procedure maintains its predictable inputs and outputs, changes to the code inside of a procedure are insulated from harming the code that calls the procedure. You can make significant changes to the procedure, but as long as the inputs and outputs are predictable and remain unchanged, the code will work just fine.

Top-Down versus Event-Driven

Before you leave this introduction to programming, it may be helpful to point out that you will encounter two different *models* of programming in this book: top-down and event-driven programs. The differences between the two have to do with the way you organize your code and how and when that code gets executed at runtime. As you get deeper into programming in general, and VBScript in particular, this will become clearer, so don’t be alarmed if it doesn’t completely sink in right now.

Understanding Top-Down Programming

So far in this chapter you've written very simple top-down style programs. The process is simple to follow:

- ❑ Write some code.
- ❑ Save the code in a script file.
- ❑ Use Windows Script Host to execute the script.
- ❑ The Script Host starts executing at the first line and continues to the last line.
- ❑ If a script file contains some procedure definitions (such as your `PromptUserName` function), then the Script Host only executes those procedures if some other code calls them.
- ❑ Once the Script Host reaches the last line of code, the lifetime of the script ends.

Top-down programs are very useful for task-oriented scripts. For example, you might write a script to search your hard drive for all the files with the extension `.htm` and copy all the names and file locations to a file, formatted in HTML to act as a sitemap. Or you might write a script that gets executed every time Windows starts and which randomly chooses a different desktop wallpaper bitmap file for that session of Windows. Top-down programming is perfect for these kinds of scripts.

Understanding Event-Driven Programming

Event-driven code is different, and is useful in different contexts. As the name implies, event-driven code only gets executed when a certain *event* occurs. Until the event occurs, the code won't get executed. If a given event does not occur during the lifetime of the script, the code associated with that event won't be executed at all. If an event occurs, and there's no code associated with that event, then the event is essentially ignored.

Event-driven programming is the predominant paradigm in Windows programming. Most of the Windows programs you use every day were written in the event-driven model. This is because of the graphical nature of Windows programs. In a graphical user interface (GUI), you have all sorts of buttons, drop-down lists, fields in which to type text, and so on. For example, the word processor program Microsoft Word is totally jam-packed with these. Every time a user clicks a button, chooses an item in a list, or types some text into a field, an event is "raised" within the code. The person who wrote the program may or may not have decided to write code in response to that event. However, if the program is well written, an item such as a button for saving a file, which the user expects to have code behind it, will indeed have code behind it.

How Top-Down and Event-Driven Work Together

When a GUI-based program starts, there is almost always some top-down style code that executes first. This code might be used to read a setting stored in the registry, prompt the user for a name and password, load a particular file at startup or prompt to take the user through setup if this is the first time the application has been run, and so on. Then a form typically comes up. The form contains all the menus, buttons, lists, and fields that make up the user interface of the program. At that point, the top-down style coding is done, and the program enters what is known as a wait state. No code is executing at this point and the program just waits for the user to do something. From here on, it's pretty much all about events.

When the user begins to do something, the program comes to life again. Suppose the user clicks a button. The program raises the `Click` event for the button that the user clicked. The code attached to that event starts to execute, performs some operations, and when it's finished, the program returns to its wait state.

As far as VBScript is concerned, the event-driven model is used heavily in scripting for the Web. Scripts that run inside of HTML web pages are all based on events. One script may execute when the page is loaded, while another script might execute when the user clicks a link or graphic. These "mini scripts" are embedded in the HTML file, and are blocked out in a syntax very similar to the one you used to define the `PromptUserName` function in the previous section.

An Event-Driven Code Example

As you progress through the second half of this book, the finer points of event-driven programming will become much clearer to you. However, just so you can see an example at this point, type the following code into your text editor, save the file with a `.HTM` extension, and then load it into Internet Explorer 6 (if you are running Internet Explorer 6/7 and you are running this file off your desktop, you might have to dismiss some security warnings and allow ActiveX).

```
<html>
<head>
<title>Simple VBScript Example</title>
<script language="vbscript">
  Sub ButtonClicked
    window.alert("You clicked on the button!")
  End Sub
</script>
</head>
<body>
  <button name="Button1" type="BUTTON" onclick="ButtonClicked">
    Click Me If You Can!!!
  </button>
</body>
</html>
```

Figure 1-8 shows the result of clicking the button on the page. In this case it's only a message box but it could be much more.

Coding Guidelines

It's a really good idea to get into healthy programming habits right from the beginning. As you continue to hone your programming skills and possibly learn multiple languages, these habits will serve you well. Your programs will be easier for you and your fellow developers to read, understand, and modify, and they will also contain fewer bugs.

When you first start writing code, you have to concentrate so hard on just getting the syntax correct for the computer that it may be easy for you to forget about all the things you need to do in order to make sure your code is human friendly as well. However, attentiveness early on will pay huge dividends in the long run.

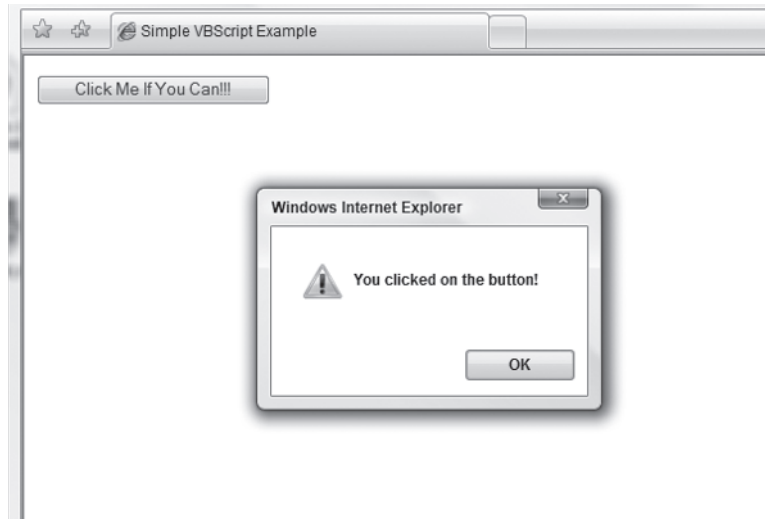


Figure 1-8

Expect the Unexpected

Always remember that anything that can happen probably will happen. The idea here is to code defensively — preparing for the unexpected. You don't need to become totally fixated on preparing for all contingencies and remote possibilities, but you can't ignore them either. You especially have to worry about the unexpected when receiving input from the user, from a database, or from a file.

Whenever you're about to perform an action on something, ask yourself questions such as:

- What could go wrong here?
- What happens if the file is flagged read-only?
- What happens if the file isn't there?
- What happens if the user doesn't run the program from the right folder?
- What happens if the database table doesn't have any records?
- What happens if the registry keys I was expecting aren't there?
- What happens if the user doesn't have the proper permission to carry out the operation?

If you don't know what might go wrong with a given operation, find out through research or trial and error. Get others to try out your code and get their feedback on how it worked for them, on their system configuration, and on their operating system. Don't leave it up to your users to discover how well (or not) your script reacts to something unexpected. A huge part of properly preparing for the unexpected is the implementation of proper error handling, which is discussed in detail in Chapter 6.

Always Favor the Explicit over the Implicit

When you are writing code, constantly ask yourself: Is my intent clear to someone reading this code? Does the code speak for itself? Is there anything mysterious here? Are there any hidden meanings? Are the variable names too similar to be confusing? Even though something is obvious in your mind at the moment you are typing the code, it doesn't mean it will be obvious to you six months or a year from now — or to someone else tomorrow. Always endeavor to make your code as self-documenting as possible, and where you fall short of that goal (which even the best programmers do — self-documenting code can be an elusive goal), use good comments to make things clearer.

Be wary of using too many generics in code, such as `x`, `y`, and `z` as variable names and `Function1`, `Function2`, and `Function3` as function names. Instead, make them explicit. Use variable names such as `UserName` and `TaxRate`. When naming a variable, use a name that will make it clear what that variable is used for. Be careful using abbreviations. Don't make variable names too short, but don't make them too long either (10–16 characters is a good length, but ideal length is largely a matter of preference). Even though VBScript is not case-sensitive, use mixed case to make it easier to distinguish multiple words within the variable name (for example, `UserName` is easier to read than `username`).

When naming procedures, try to choose a name that describes exactly what the procedure does. If the procedure is a function that returns a value, indicate what the return value is in the function name (for example, `PromptUserName`). Try to use good verb–noun combinations to describe first, what action the procedure performs, and second, what the action is performed on (for example, `SearchFolders`, `MakeUniqueRegistryKey`, or `LoadSettings`).

Good procedure names tend to be longer than good variable names. Don't go out of your way to make them longer, but don't be afraid to either. Fifteen to thirty characters for a procedure name is perfectly acceptable (they can be a bit longer because you generally don't type them nearly as much). If you are having trouble giving your procedure a good name, that might be an indication that the procedure is not narrow enough — a good procedure does one thing, and does it well.

That said, if you are writing scripts for web pages to be downloaded to a user's browser, it is sometimes necessary to use shorter variable and procedure names. Longer names mean larger files to download. Even if you sacrifice some readability to make the file smaller, you can still take time to create descriptive names. With web scripts, however, you may encounter instances where you don't want the code to be clear and easy to understand (at least for others). You'll look at techniques that you can employ to make scripts harder for “script snoopers” to follow while still allowing you to work with them and modify them later (see Chapter 17).

Modularize Your Code into Procedures, Modules, Classes, and Components

As you write code, you should constantly evaluate whether any given code block would be better if you moved it to its own function or subprocedure:

- Is the code rather complex? If so, break it into procedures.
- Are you using many `And`s and `Or`s in an `If . . . End If` statement? Consider moving the evaluation to its own procedure.

Chapter 1: A Quick Introduction to Programming

- ❑ Are you writing a block of code that you think you might need again in some other part of the script, or in another script? Move it to its own procedure.
- ❑ Are you writing some code that you think someone else might find useful? Move it.

This isn't a science and there are no hard and fast rules for code — after all, only you know what you want it to do. Only you know if parts are going to be reused later. Only you know how complex something will turn out. However, always keep an eye out for possible modularization.

Use the “Hungarian” Variable Naming Convention

You might hear programmers (especially C++ programmers) mention this quite a bit. While this is a bit out of scope of this introductory discussion, it is still worth mentioning nonetheless. The Hungarian naming convention involves giving variable names a prefix that indicates what the scope and data type of the variable are intended to be. So as not to confuse matters, the Hungarian convention was not used in this chapter, but you will find that most programmers prefer this convention. Properly used, it makes your programs much clearer and easier to write and read.

See Chapter 3 for more on Hungarian notation variable prefixes. The standard prefixes for scope and data types are in Appendix B.

Don't Use One Variable for More Than One Job

This is a big no-no and a common mistake of both beginner and experienced programmers alike (but the fact that experienced programmers might have a bad habit does not make it any less bad). Each variable in your script should have just one purpose.

It might be very tempting to just declare a bunch of generic variables with fuzzy names at the beginning of your script, and then use them for multiple purposes throughout your script — but don't do it. This is one of the best ways to introduce very strange, hard to track down bugs into your scripts. Giving a variable a good name that clearly defines its purpose will help prevent you from using it for multiple purposes. The moral here is that while reusing variables might seem like a total timesaver, it isn't and can lead to hours of frustration and wasted time looking for the problem.

Always Lay Out Your Code Properly

Always remember that good code layout adds greatly to readability later. Don't be tempted to save time early on by writing messy, hard to follow code because as sure as day turns to night, you will suffer if you do.

Without reading a single word, you should be able to look at the indentations of the lines to see which ones are subordinate to others. Keep related code together by keeping them on consecutive lines. Also, don't be frightened of white space in your code. Separate blocks of unrelated code by putting a blank line between them. Even though the script engine will let you, avoid putting multiple statements on the same line. Also, remember to use the line continuation character (`\`) to break long lines into multiple shorter lines.

The importance of a clean layout that visually suggests the logic of the underlying code cannot be overemphasized.

Use Comments to Make Your Code More Clear and Readable, but Don't Overuse Them

When writing code, strive to make it as self-documenting as possible. You can do this by following the guidelines set out earlier. However, self-documenting code is hard to achieve and no one is capable of 100% self-documenting code. Everyone writes code that can benefit from a few little scribbles to serve as reminders in the margins. The coding equivalents of these scribbles are comments. But how can you tell a good comment from a bad comment?

Generally speaking, a good comment operates at the level of intent. A good comment answers the questions:

- ❑ Where does this code block fit in with the overall script?
- ❑ Why did the programmer write this code?

The answers to these questions fill in the blanks that can never be filled by even the best, most pedantic self-documenting code. Good comments are also generally “paragraph-level” comments. Your code should be clear enough that you do not need a comment for each and every line of code it contains, but a comment that quickly and clearly describes the purpose for a block of code allows a reader to scan through the comments rather than reading every line of code. The idea is to keep the person who might be reading your code from having to pore over every line to try and figure out why the code exists. Commenting every line (as you probably noticed with the earlier examples) makes the code hard to follow and breaks up the flow too much.

Bad comments are generally redundant comments, meaning they repeat what the code itself already tells you. Try to make your code as clear as possible so that you don't need to repeat yourself with comments. Redundant comments tend to add clutter and do more harm than good. Reading the code tells you the how; reading the comments should tell you the why.

Finally, it's a good idea to get into the habit of adding “tombstone” or “flower box” comments at the top of each script file, module, class, and procedure. These comments typically describe the purpose of the code, the date it was written, the original author, and a log of modifications.

```
' Kathie Kingsley-Hughes
' 22 Feb 2007
' This script prompts the user for his or her name.
' It incorporates various greetings depending on input by the user.
'
' Added alternative greeting
' Changed variable names to make them more readable
```

Summary

In this chapter you took a really fast-paced journey through the basics of programming. The authors tried to distill a whole subject (at least a book) into one chapter. You covered an awful lot of ground but also skimmed over or totally passed by a lot of stuff. However, the information in this chapter gave you the basics you need to get started programming with VBScript and the knowledge and confidence you need to talk about programming with other programmers in a language they understand.

